自制搜索引擎

How to Develop a Search Engine

[日] 山田浩之　末永匡 / 著　胡屹 / 译

TURING
图灵程序设计丛书

2600行代码
真实体验搜索引擎的开发过程

开源搜索引擎Senna/Groonga的开发者亲自执笔

Google、百度的工作机制

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：自制搜索引擎

作者：［日］ 山田浩之　末永匡

译者：胡屹

ISBN：978-7-115-41170-9

版权声明
声明
译者序
前言
第 1 章　搜索引擎是如何工作的
1-1　理解搜索引擎的构成
什么是搜索引擎
构成搜索引擎的组件
与搜索引擎相关的组件
1-2　实现了快速全文搜索的索引结构
全文搜索的两种方法
倒排索引的结构
倒排索引的构建方法
倒排索引中的术语
1-3　深入理解倒排索引
倒排索引= 词典+ 倒排文件
从倒排索引中查找单词
将单词的位置信息加入倒排文件中
从倒排索引中查找短语
1-4　制作中文文档的倒排索引
分割中文句子的方法
权衡分割方法
1-5　实现倒排索引
实现词典
实现倒排文件
1-6　使用倒排索引进行检索
布尔检索
使用倒排索引的检索处理流程

关联度的计算方法
信息检索中的检索
1-7　构建倒排索引
使用内存构建倒排索引
使用二级存储构建倒排索引
静态索引构建和动态索引构建
1-8　准备要检索的文档
收集数据
数据规范化
第 2 章　准备全文搜索引擎的检索样本
2-1　全文搜索引擎 wiser
wiser 的构成
准备用于检索的文档
2-2　安装 wiser
构建 wiser
启动 wiser
解压缩 Wikipedia 的副本
2-3　运行 wiser
构建倒排索引
使用倒排索引查询
比较 grep 和 wiser 的运行速度
第 3 章　构建倒排索引
3-1　复习有关倒排索引的知识
提取词元
为每个词元创建倒排列表
3-2　构建倒排索引
在存储器上创建倒排列表
倒排列表和倒排文件的数据结构

从源代码级别梳理倒排索引的构建顺序
进一步阅读源代码
第 4 章　开始检索吧
4-1　检索处理的大致流程
充分理解检索处理的流程
4-2　使用倒排索引进行检索
从源代码级别梳理检索处理的流程
解读 split_query_to_tokens() 函数的具体实现
使用具体示例加深对检索处理流程的理解
解读函数 search_docs() 的实现细节
解读函数 search_phrase() 的实现
第 5 章　压缩倒排索引
5-1　压缩的基础知识
压缩倒排索引的好处
倒排索引的压缩方法
倒排文件的压缩方法
压缩的原理
5-2　实现wiser 中的压缩功能
压缩功能源代码的概要
了解无需进行压缩时的操作
抓住 Golomb 编码的要点
解读 Golomb 编码中的编码处理
解读 Golomb 编码的解码处理
第 6 章　挑战wiser的优化及参数的调整
6-1　提高检索处理的效率
优化检索处理
将查询分割为无重复部分的词元序列
6-2　禁用短语检索

分析对 2 字符的字符串进行检索时的行为
分析对 3 字符的字符串进行检索时的行为
6-3　改变检索结果的输出顺序
作为检索结果排序核心的指标
按照文档大小降序排列的检索结果
6-4　让1 个字符的查询也能检索出结果
获取以特定字符开头的词元的列表
合并检索到的结果
6-5　调整控制倒排索引更新的缓冲区容量
确认由缓冲区容量的差异带来的不同效果
用 sar 命令分析负载
6-6　调整只有英文字母的词元的分割方法
如何避免用英文单词检索时准确率下降的问题
如何判断某字符是否属于索引对象
修改负责分割词元的函数
6-7　确认压缩的效果
观察Golomb 编码的效果
对比压缩启用前后的索引大小
第 7 章　为今后更加深入的学习做准备
7-1　wiser 没能实现的功能
倒排索引之外的全文搜索索引
高效处理大规模数据的存储器
利用缓存提高检索的速度
使用各种各样的压缩方法
优化搜索结果的排名
调整准确率和召回率
降低检索结果排序处理的负载
并行处理

结合对属性的筛选过滤
分面搜索
7-2　全文搜索引擎 Groonga 的特点
通过词元的部分一致检索提升召回率
使用内存映射文件
片段
7-3　实现出考虑到用户意图的搜索引擎
引入停用词
应对词素解析的错误
处理全角字符和半角字符
对查询进行归一化
留意布尔检索的解析过程
通过词素解析器适当地解析查询
对错误的输入进行修正
输入补全
建议用户检索相关的关键词
7-4　收集、提取文档时的要点
制作爬虫时的处理要点
在提取文本时需要处理的要点
附录
A-1　深度话题
近几年的压缩方法
动态索引构建
分布式索引
A-2　wiser 中的文本提取和存储
用于处理 XML 的 2 种 API——DOM 和 SAX
提取文档的标题和正文
掌握状态的迁移

构建文档数据库
后记

版权声明

KENSAKU ENGINE JISAKU NYUMON by Hiroyuki Yamada, Tasuku Suenaga

Original Japanese edition published by Gijyutsu-Hyoron Co., Ltd., Tokyo

This Simplified Chinese language edition published by arrangement with Gijyutsu-Hyoron Co., Ltd., Tokyo in care of Tuttle-Mori Agency, Inc., Tokyo

本书中文简体字版由 Gijyutsu-Hyoron Co., Ltd. 授权人民邮电出版社独家出版。

声明

- 免责声明

  本书所述内容仅以提供信息为目的。因此，请读者务必根据自身的义务和判断运用书中的信息，由此产生的任何后果，技术评论社、原书作者、人民邮电出版社以及译者概不负责。

  由于本书所提供的是截至 2014 年 8 月 8 日的信息，所以在使用时亦会出现所述内容已发生变更的情况。

  另外，软件版本的升级会导致其功能和界面等与本书所述不符。因此，在购买本书前，请您务必确认软件的版本号。

  请在阅读并同意了上述注意事项之后再使用本书，否则，技术评论社、原书作者、人民邮电出版社以及译者恐怕难以回复您的询问。这一点还望诸位读者事先知晓。

- 关于商标和注册商标

  本书所述的产品名称多为各相关公司的商标或注册商标。另外，本书已省略了 ™、® 等符号。

译者序

《自制搜索引擎》一书终于和读者们见面了，“自制”系列图书的家族中又多了一名新成员。近几年，图灵先后出版了几本“自制”系列图书，如《30 天自制操作系统》《自制编程语言》《两周自制脚本语言》等。在这些书中，我们不用去读枯燥乏味的原理和晦涩难懂的算法，只需跟随作者的脚步，即可从零开始，一步步地创造出操作系统或编程语言的雏形。

《自制搜索引擎》一书也不例外。在这本不到 200 页的书中，作者先用简明扼要、通俗易懂的语言为我们讲解了搜索引擎的结构及核心概念，紧接着又带领我们剖析了一个名为 wiser 的原创搜索引擎的源代码。理论与大量源代码的结合帮助我们迈入了搜索引擎的大门，只要用心阅读并实际操作，就能制作出一个可以在计算机上运行的简易搜索引擎。然而与其他计算机技术一样，虽然搜索引擎的入门很简单，但要成为这个领域的技术专家却并不容易，离不开大量的知识积累和实践。所以在分析完源代码以后，作者又带领我们优化了现有的 wiser 搜索引擎，并简单地介绍了一些更加专业的知识，以启发我们深入思考，为进一步学习铺平了道路。

阅读本书几乎不需要任何有关搜索引擎的知识储备，但由于 wiser 是用 C 语言编写的，所以您最好还是能有些 C 语言的编程经验。“啊，用 C 写的啊？”也许您也和我当初一样，一听是 C 语言就泄气了。的确，C 语言不是那么好用。指针是个难点不说，有些语句的写法也显得很诡异，而且还缺乏丰富的内置函数和数据结构。但如果您坚信某某语言才是世界上最好的语言，并要因此放弃本书的话，那么我建议您先下载 wiser 的源代码读一读再做决定。wiser 的源代码仅有大约 2600 行。即使只瞥一眼，也应该能够发现这些源代码不但具有详细的注释、清晰的结构，而且遵循了良好的命名规范。仔细地阅读后，甚至还能看到有些地方应用了回调函数、设计模式等所谓的“现代”编程技巧。不仅如此，作者还通过引入了名为 uthash 的代码库简化了对字符串、列表和哈希表的操作。例如要向列表中添加元素时，只需使用形如“LL_APPEND(*list, element);”的一行代码，这就大大增加了代码的可读性。相信您读到最后也会由衷地感叹：原来 C 语言也能这么好用啊。

对于想要开发搜索引擎的读者来说，本书的作用自不必说。而对于专注于其他领域的开发者，甚至对于那些只是想学门新技术来娱乐一下的程序员来说，读读本书也是大有裨益的。例如，我们可以从中学到如何高效地求得多个大集合的交集，如何压缩存储大量的整数，如何运用 sar 命令查看并分析系统的性能等。即使我们不从事搜索引擎的开发工作，这些算法和技术也会对日常的工作有所启发和帮助。所以，读过了本书，就算您并不打算做一个搜索引擎出来，也能得到一些收获。

值得一提的是，在本书中很多叙述得较为简练甚至一笔带过的段落中，其实隐藏着大量的知识。在掌握了搜索引擎的核心技术后，不妨查查资料、写写代码，试着去掌握这些更高级的知识，搞清楚里面专业术语的含义。例如，书中提到了字典树（Tier）、Suffix Array 等国内教材中罕见的数据结构，那么我们能不能用自己熟悉的编程语言实现它们？作者开发的开源搜索引擎 Groonga 采用了内存映射文件技术，那么内存映射文件的机制是什么……在不断探索这些问题的过程中，我们不但能把这本不算厚的书读得越来越厚，也能使自己的知识量不断增长。

最后，在这里衷心感谢在翻译过程中给予我支持与鼓励的各位。欢迎诸位读者批评指正，提出宝贵的建议。希望所有对搜索引擎感兴趣的读者都能从本书中获益。

胡屹

2015 年 10 月于北京

前言

本书聚焦于 Google 和 Yahoo! 等 Web 检索服务幕后的搜索引擎，旨在阐明这种系统内部的工作机制。诸位读者通过第 1 章的学习，掌握了搜索引擎的基础知识和原理之后，就可以从第 2 章开始，对照着示例搜索引擎的源代码体验搜索引擎的开发过程了。这种原理和实践的有机结合，有助于大家更加深入地理解搜索引擎的构造。

一直在企业和大学从事搜索引擎研发工作的山田负责搜索引擎原理的写作，并完成了整体构思和统稿的工作。开源搜索引擎 Senna/Groonga 的开发者、拥有多个检索服务实战经验的末永在书中介绍了实践和运用搜索引擎时的要点。这种内容上的相互补充使得原理和实践有机地结合在了一起。

若从本书获得的知识和经验能有助于诸位读者创造出划时代的软件和服务，我们将感到不胜荣幸。

山田浩之

于 2014 年 8 月

第 **1** 章　搜索引擎是如何工作的

在体验搜索引擎的开发过程之前，我们先在第 1 章介绍一下搜索引擎的基本概念。搜索引擎的基础是应用于信息检索、数据库等领域的信息技术，要想开发搜索引擎，横跨多个领域的广泛知识是不可或缺的。在本章我们尽可能通俗易懂、简明扼要地总结归纳了这些知识。由于本章讲解的是后续章节的背景知识，所以恳请诸位认真地读下去。

1-1　理解搜索引擎的构成

在本节，我们首先介绍什么是搜索引擎，然后再大略地讲解其基本架构。由于从 1-2 节开始还会详细地讲解有关内容，所以在本节就让我们先在大体上了解一下搜索引擎的全貌吧。

什么是搜索引擎

搜索引擎是一类系统或软件的统称，作用是从文档的集合中查找（检索）出匹配信息需求（查询）的文档，信息需求是由单词、问题等构成的。

确切地说，本书所讲解的搜索引擎其实是“全文搜索引擎”。所谓的“全文”指的就是全部的句子，当检索的对象为“由文本构成的文档中的全部句子”时，对于该文档进行的检索就称为全文搜索。而实现了这种全文搜索的系统就是全文搜索引擎（全文搜索系统），在英文中一般称为 Full-text Search Engine。在本书之后的章节中，提到“搜索引擎”指的就是全文搜索引擎。

在现代的搜索引擎中，不仅能看到 Google 和 Yahoo! 等 Web 检索，还可以看到邮件检索和专利检索等各式各样的应用程序（应用层）。当然，应用程序的用途和使用方式不同，搜索引擎的规模和其所要求的系统必备条件也就不同。尽管如此，在这些应用程序中，搜索引擎的基本结构却没有太大的差异。本书将以搜索引擎的基本结构为主进行讲解。

下面，就让我们先从搜索引擎的全貌看起吧。

构成搜索引擎的组件

搜索引擎一般由以下 4 个组件构成。

- 索引管理器（**Index Manager**）
- 索引检索器（**Index Searcher**）
- 索引构建器（**Indexer**）

- 文档管理器（**Document Manager**）

图 1-1 展示了构成搜索引擎的全部要素。首先让我们简单地看看这些组件都在进行着怎样的工作吧。

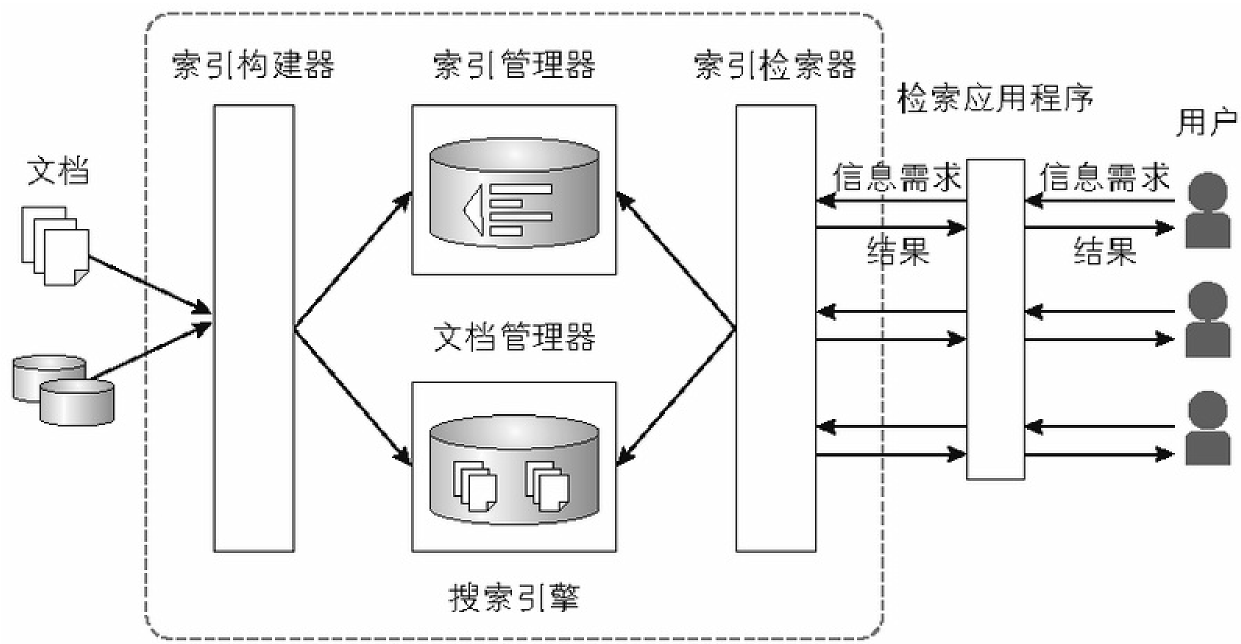


图 **1-1**　搜索引擎的构成

▌索引管理器

索引管理器组件的作用是管理带有索引结构的数据，索引结构是一种用于进行高速检索的数据结构。对索引的访问也是通过索引管理器进行的。

索引管理器通常是将索引作为二级存储上的二进制文件来进行管理的。而且，还经常会通过保存经过压缩的索引来达到减少从二级存储加载的数据量，提升检索处理效率的目的。

▌索引检索器

索引检索器是利用索引进行全文搜索处理的组件。索引检索器根据来自检索应用程序用户的查询，协同索引管理器进行检索处理。在大多数情况下，索引检索器都会根据某种标准对与查询相匹配的检索结果排序，并将排在前面的结果返回给应用程序。

另外，本书将查询和信息需求视为同义词。所谓查询是指“由 1 个以上的单词或词组组成的对搜索引擎的询问”。

▍索引构建器

索引构建器是从作为检索对象的文本文档中生成索引的组件。索引构建器会先通过解析将文本文档分解为单词序列，然后再将该单词序列转换为索引结构。在搜索引擎中，将生成索引的环节称为索引构建(Index Construction)。

▍文档管理器

文档管理器是管理文档数据库的组件，文档数据库中储存着作为检索对象的文档。文档管理器会先从文档数据库中取出与查询相匹配的文档，然后再根据需要从该文档中提取出一部分内容作为摘要。

由于文档管理器的结构非常简单，只是对应着文档特定的 ID（文档编号）来保存文档的内容，所以本书就省略了相关的详细介绍。我们经常能看到有人将数据库管理系统（DBMS）和基于二级存储的数据库管理器（DBM）等用作文档管理器。

由文档管理器管理的文档数据库既可以在构建索引的阶段随索引一同构建，也可以提前构建。

与搜索引擎相关的组件

严格来讲，本节所介绍的爬虫和搜索排序系统虽不是搜索引擎的一部份，但却是与搜索引擎密切相关的组件。

▍爬虫

爬虫 (Crawler) 是用于收集 Web 上的 HTML 文件等文档的系统（机器人）。例如，用于 Web 检索的爬虫就是通过追随 Web 页面上的超链接来收集全世界的 HTML 网页的。全世界的 Web 页面正以惊人的速度不断增长，因此爬虫的任务就是高效地收集这些网页。

▍搜索排序系统

以 Google 的 PageRank 系统为代表的搜索排序系统是给作为检索对象的文档打分的系统。例如，在 Web 检索中，通常会以考量了查询与文档的关联性以及文档的热门度后得出的分数为基准，将检索结果排序后提供给应用程序的用户。搜索排序系统正是用于此目的的、能（机械地）算出文档热门度的系统。

在本节，我们讲解了搜索引擎的一般构成以及各个组件的主要用途。由于在后面的章节还会继续一一讲解各个组件，所以即使现在还未能充分理解也不必担心，可以先从大体上把握搜索引擎的全貌。

1-2　实现了快速全文搜索的索引结构

本节讲解的是用于快速进行全文搜索的索引结构。在讲解广泛应用于全文搜索的、名为倒排索引的索引结构之前，让我们先来梳理一下全文搜索的方法。

全文搜索的两种方法

全文搜索大致上可以分为两种方法，一种是利用全扫描进行全文搜索，一种是利用索引进行全文搜索。

❚ 利用全扫描进行全文搜索

第一种方法是从头到尾扫描作为检索对象的文档，以此来搜索要检索的字符串。由于 Unix 的字符串检索命令“grep”也是以同样的方式进行搜索的，所以有时也将这种方法称为“grep 型搜索”。

在利用全扫描进行全文搜索时，虽然不需要事先处理作为检索对象的文档，但问题是文档数越多检索时间就越长。因此，一般认为这种方法只适用于处理少量或暂时性的文档。

另外，在通过对文档进行全扫描来搜索字符串的方法中，有一些高效的算法，例如 KMP 算法和 BM 算法。本书并不会介绍这些算法，若诸位有兴趣的话可以去参考有关算法的教材。

❚ 利用索引进行全文搜索

相对于利用全扫描进行全文搜索的方法，第二种方法，即利用索引的方法，则需要事先为文档建立索引，然后利用索引来搜索要检索的字符串。虽然事先建立索引需要花费时间，但是优点是即使文档的数量增加，检索速度也不会大幅下降。因此，一般认为这种方法更适合处理大量的文档。搜索引擎一般也会采用这种方法。

虽然索引分为很多种，每种的结构都不同，但是以 Google 和 Yahoo! 为代表的大多数搜索引擎采用的都是名为倒排索引的索引结构。也就是说，在全文搜索中倒排索引是一种最常见的索引结构。各位将要通过本

书体验其开发过程的搜索引擎采用的也是倒排索引。

下面我们就开始讲解倒排索引的结构。

倒排索引的结构

虽然看似与本节的主题无关，但还是请诸位先回想一下印在教材或专业书等图书后面的索引。在书后的索引中，通常都会写有关键词（单词）和出现了该关键词的页码。由于关键词是按词典顺序排列的，所以查找时无需逐一浏览，只需按照拼音字母的顺序逐渐缩小查找范围，就能轻松地找到关键词。而只要再向这个关键词的旁边看一眼，就能立刻知道该关键词出现在哪一页了。

实际上，倒排索引具有与图书索引完全相同的逻辑结构。下面就让我们以一本书中的文档为例来具体看看倒排索引吧。这本书由以下两页组成，内容分别如下所示。

- 第 **1** 页（**P1**）：**I like search engines.**

- 第 **2** 页（**P2**）：**I search keywords in Google.**

表 1-1 列出了这本书的倒排索引。

表 **1-1**　示例书籍中的倒排索引

engine	P1
Google	P2
I	P1,P2
in	P2
keyword	P2
like	P1
search	P1, P2

从表 1-1 应该就能看出倒排索引确实和图书的索引拥有相同的结构。看到单词时只要查一下这张表，该单词出现在哪一页就一目了然了。所谓倒排索引就是一张列出了“哪个单词出现在了哪一页”的表格。

倒排索引的构建方法

如何才能构建出倒排索引呢？下面就让我们使用上面的书籍示例，具体地看一看构建倒排索引的步骤吧。首先，要以表格的形式归纳出书中的每一页都使用了哪些单词。归纳出的表格如表 1-2 所示。请注意此时要将英文单词的复数形式还原为单数形式。

表 **1-2**　表中列出了书中的哪一页使用了哪个单词

	I	**like**	**search**	**engine**	**keyword**	**in**	**Google**
P1	1	1	1	1	0	0	0
P2	1	0	1	0	1	1	1

在表 1-2 中，我们以书中使用过的单词为行标题，以页码为列标题。写好行、列标题后，当某页使用了某个单词，就将 1 填入对应的格子中。例如，第 1 页（P1）使用了 I、like、search、engine 几个单词，因此就在对应的几个格子中写入 1。对于第 2 页（P2）也是如此。由于表格是从左向右填写的，所以自然也要从左向右浏览，这样就能读出出现在各页中的单词了。若是从上向下浏览，又会有什么发现呢？这样浏览应该能读出每个单词都出现在哪一页上了。为了便于浏览，我们交换了表 1-2 的行和列，并将单词按照词典顺序进行了排序，最终结果如表 1-3 所示。

表 **1-3**　表中列出了书中的哪个单词出现在了哪一页上

	P1	**P2**
engine	1	0
Google	0	1

I	1	1
in	0	1
keyword	0	1
like	1	0
search	1	1

从表 1-3 应该就能看出，实际上到了这个阶段，倒排索引就已经大致完成了。剩下的步骤就与制作图书的索引一样了，只要改用精简的表示方式，即只列出“每个单词都出现在了第几页上”，就可以制成表 1-1 那样的表格了 [1]。

> [1]实际上也有用 0/1 填充表格的表示方法。这是一种称为位图（Bitmap）的有别于倒排索引的表示方法。在处理大量的文档时，位图的表会变得非常大，而且表中的大部分元素都是 0（成为了稀疏表），这导致表虽然很大，但是信息量却很少（没有有效地利用空间），所以现在已经很少使用位图作为检索的索引结构了。

像这样，将表示“在哪一页上使用了哪个单词”的表格转换为“哪个单词出现在了哪一页上”的表格，就可以制作出倒排索引了。另外，之所以称这种索引为倒排索引，是由于刚刚进行过的交换表格行、列的操作叫作“倒排”。

倒排索引中的术语

在生成的倒排索引中，我们建立起了页中的单词和页的对应关系。也就是说，我们是把页当成了构建索引的单位。之所以这样做，是因为在翻阅图书时，人们通常是以“页”作为单位的。

那么，对于其他情况又要如何处理呢？例如，对于 Web 上的 HTML 文档，我们可以将 1 个 HTML 网页作为构建索引的单位。而对于邮件，我们可以将 1 封邮件作为构建索引的单位。

由此可见，对于每种作为检索对象的数据，构建索引的单位都是不同的。在全文搜索中，将构建索引的单位统称为“文档”（Document），将文档的标识信息称为“文档编号”。文档编号类似图书的页码，用于唯一地标识某个文档。因此，也可以这样说，所谓倒排索引就是“把单词和单词所在文档的文档编号对应起来的表格”。

另外，在倒排索引中，将表示单词和文档编号对应关系的信息称为倒排项（Posting），将各个单词的倒排项的集合称为倒排列表（Postings List）。例如，在刚刚的倒排索引中，单词 search 的倒排项是 P1 和 P2，倒排列表是 P1、P2 的集合，记作“P1,P2”。

1-3　深入理解倒排索引

至此为止，我们就讲解完了倒排索引的概要。下面，就让我们再来略微详细地了解一下倒排索引吧。

倒排索引= 词典+ 倒排文件

倒排索引是由单词的集合“词典”和倒排列表的集合“倒排文件”构成的。词典和倒排文件以及作为这二者构成要素的单词和倒排列表的关系如图 1-2 所示。

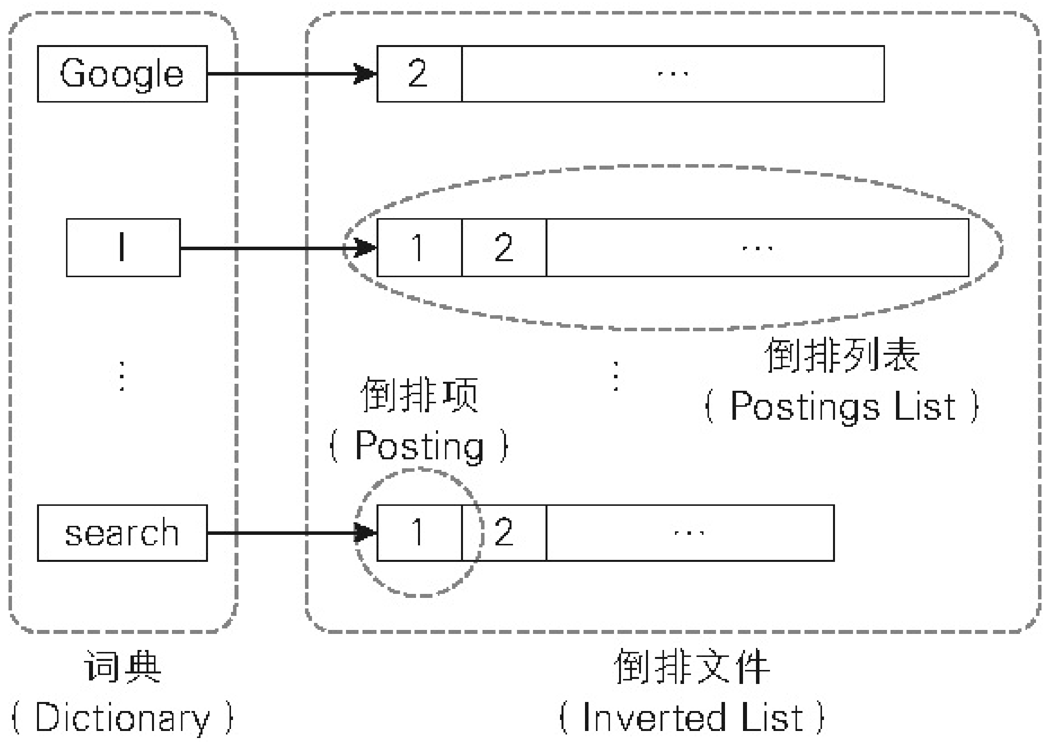


图 **1-2**　倒排索引的结构

词典中的每个单词都持有一段引用信息，指明了对应着该单词的倒排列表。利用这段引用信息，我们就可以从词典中的各个单词那里获取到相应的倒排列表了。

从倒排索引中查找单词

若要从倒排索引中查找出包含了某个单词的文档，只需要先从词典中找到该单词，然后获取与之对应的倒排列表，最后从倒排列表中获取文档编号即可。这里只是改用检索的术语将之前讲解过的方法描述了一遍，所以若有疑问的话，请再重新读读前面的内容。

那么，我们又该如何查找同时包含了多个单词的文档呢？查找时只需要先从词典中找出各个单词，然后分别获取这些单词的倒排列表并加在一起，由此计算出包含在各个倒排列表中的文档编号的交集。举例来说，假设我们使用的是上一节生成的那个倒排索引，并要从中查找出既包含 search 又包含 engine 的文档。那么根据上述方法，一旦获取到了 search 和 engine 分别对应的倒排列表，就可以知道 search 包含在页面 1（P1）和页面 2（P2）中，engine 包含在页面 1（P1）中。而接下来，只要再计算出这两个倒排列表的交集，就又可以知道同时包含这两个单词的文档是文档 1（P1）。

将单词的位置信息加入倒排文件中

到目前为止，我们见到的倒排文件都只带有“各单词都出现在了哪个文档中”这一种信息。这样的倒排文件称为“文档级别的倒排文件”（Document-level Inverted File）。

除此以外，还有另一种倒排文件，称作“单词级别的倒排文件”（Word-level Inverted File）。这种倒排文件中不仅带有有关单词出现在了哪个文档中的信息，还带有单词出现在了文档中的什么位置（从开头数是第几个单词）这一信息。

在单词级别的倒排文件中，各个倒排项的表示方法如下所示。

DocID;offset1, offset2...

还是以刚刚使用过的两个文档为例，从头数的话，单词 search 是文档 1（P1）中的第 3 个单词，是文档 2（P2）中的第 2 个单词，因此其倒排列表是

search: D1;3, D2;2

如果把各个单词在文档中的出现位置都如此考察一遍，就可以得到如下所示的单词级别的倒排文件了。

engine: D1;4

Google: D2;5

I: D1;1,D2;1

in: D2;4

keyword: D2;3

like: D1;2

search: D1;3, D2;2

随后我们要介绍的从倒排索引中查找短语，或是计算检索结果中文档的得分等场景中都会用到这种单词的位置信息。

从倒排索引中查找短语

我们刚刚讲解的是如何利用文档级别的倒排文件查找同时包含 search 和 engine 的文档。但是利用这种方法得到的检索结果，未必都是关于搜索引擎（search engine）的文档。例如，虽然下面的文档也同样包含了 search 和 engine，但却与搜索引擎无关。

I search for a gas station because my car's engine doesn't start.

（因为汽车的引擎发动不起来了，所以我要找加油站。）

因此，要想查找关于搜索引擎的文档，就需要从倒排索引中找出含有短语 search engine 的文档。而要想从倒排索引中查找短语，就需要使用刚刚介绍过的单词级别的倒排文件 [2]。

> [2]也可以使用文档级别的倒排文件找出含有短语 search engine 的文档，方法是在检索完各个单词之后，用全扫描的方式在原文档中检索该短语。但是，这样做的效率通常较低。

在使用单词级别的倒排文件查找短语时，前几步与使用文档级别的倒排列表相同，即也是先从词典中找出单词 search 和 engine，然后分别获取它们的倒排列表，最后算出这两个倒排列表中文档编号的交集。但 是到这里还没有结束，查找短语时还需要确认 search 和 engine 是否是相

邻出现的。在上面的例子中，由于 search 和 engine 都出现在了文档 1 中，并且 search 是文档 1 中的第 3 个单词，engine 是第 4 个单词，这说明这两个单词是相邻出现的，所以可以得出结论，短语 search engine 出现在了文档 1 中。

1-4　制作中文文档的倒排索引

至此为止，我们就讲解完了针对英文文档的倒排索引。在英文的句子中，由于单词之间留有空白，所以通过用空白划分句子就可以提取出句中的单词。但是在中文的句子中，由于各单词是词间不留空白连续书写的，所以就需要使用不同于英文的方法，才能将句子分割成单词或字符的序列。在本节我们详细地看一下分割中文句子的方法。

分割中文句子的方法

若要将类似中文的句子，即单词无法通过空白划分出来的句子分割成单词序列，通常有以下两种方法。

- 词素解析分割法
- **N-gram**（**q-gram**）分割法

下面就让我们详细地看一看这两种分割方法吧。

▌词素解析分割法

词素解析（Morphological Analysis）分割法是一种将句子分割为“词素”序列的方法。词素是语言中含有意义的最小单位。例如，如果使用词素解析分割法分割“全文搜索引擎”这段文本，那么可以得到如下结果。

全文　搜索　引擎

由于中文的语法极其复杂，所以一般认为对中文句子正确地进行词素解析是件非常困难的事。近几年，在词素解析上，一般采用的是机器学习的方法。机器学习的过程是先学习由手工作业正确分割句子后得到的数据，然后推理出应该如何分割未知的句子（以及如何标注词性等）[3]。一般认为现代词素解析的精度已经非常高了，在大多数情况下，都能正确地判断出中文句子应该在哪里分割成词。不过，对于那种在博客等环境中常用的含有大量口语表达的句子，精度还是会大幅下降的。

[3]机器学习中的有些方法采用了隐马尔可夫模型（Hidden Markov Model），有些采用了条件随

> 机场（Conditional Random Field）概率模型。

N-gram（**q-gram**）分割法

N-gram 分割法是一种将句子分割成由 N 个字符组成的片段序列的方法，每个片段称作一个 N-gram。N 的取值通常为 2 或 3。$N=1$ 时称作 uni-gram（一元 gram），$N=2$ 时称作 bi-gram（二元 gram），$N=3$ 时称作 tri-gram（三元 gram）。例如，如果使用 bi-gram 去分割“全文搜索引擎”这段文本，那么可以得到如下结果。

全文　文搜　搜索　索引　引擎

N-gram 分割法作为一种不依赖具体语言的句子分割方法，广泛应用于以亚洲国家的语言为主的各种语言中。

另外，在英文中将分割句子这种行为称为 Segmentation 或 Tokenization。严格地来讲这两个词的含义并不相同，但是在有关倒排索引的上下文中，人们似乎并不怎么在使用上对它们加以区分。句子分割后产生的一个个单词则称为词元（Token）或词项（Term）。

权衡分割方法

上述两种方法都可以将句子分割成词元，由这两种词元构成的倒排索引各有各的优缺点。

由词素构成的倒排索引的优缺点

与由 N-gram 构成的倒排索引相比，由词素构成的倒排索引由于从文中分割出的词元数更少，所以词典和倒排文件的尺寸也就更小。由此就产生了高速进行构建处理和搜索处理的可能性。

不过这种倒排索引也存在缺点，即会发生所谓的“检索遗漏”问题。检索遗漏指的是，尽管查询实际就包含在文档中，但就是找不到与查询相匹配的内容。这是由查询与通过词素解析从文中分割出的词素不一致导致的。例如，将“哆哆嗦嗦”分割成词素后还是“哆哆嗦嗦”，可如果检索的是“哆嗦”，就无法检索到包含“哆哆嗦嗦”的文档了。那些尚未收录在词素解析词典中的新词和以口语方式使用的单词也都面临同样的问题。

由 **N-gram** 构成的倒排索引的优缺点

与由词素构成的倒排索引不同，由 N-gram 构成的倒排索引不会产生检索遗漏问题。也就是说，在由 N-gram 构成的倒排索引中，基本上只要查询包含在文档中，就一定能找得到[4]。

> [4]要想能够检索到少于 N 个字符的字符串，通常需要事先制作由 M-gram（$M < N$）构成的倒排索引。

但是，从刚刚的“全文搜索引擎”的例子中也能看出来，相比于词素解析，在同一个文档中使用 N-gram 产生的词元通常较多。因此，词典和倒排文件的尺寸自然也就更大，从而导致构建处理和搜索处理的速度下降。而且，由于 N-gram 并不考虑单词的界限，所以在由 N-gram 构成的倒排索引中，会发生检索“华山”，却也能找到包含“九华山”的文档这样的问题。

在开发搜索引擎的过程中，重要的是根据文档的特性灵活运用这两种分割方法。为了体现出运用上的灵活性，我们需要设计出不依赖句子分割方法的搜索引擎。例如，可以不以从文档的开头数是第几个词元为基础，而是以从文档的开头数是第几个字符为基础来构建倒排项。

1-5　实现倒排索引

至此为止，我们已经逐渐了解了倒排索引的逻辑结构。下面，就让我们再来了解一下倒排索引的具体实现吧。

实现词典

在实现词典时，为了能够快速地获取到对应着单词的倒排列表，通常都会使用哈希表、树等数据结构。例如，常用的树形数据结构有保存着各个单词顺序关系的二叉查找树（Binary Search Tree）和字典树（Trie）等。

▌用二叉查找树实现词典

使用二叉查找树实现词典时，要先将数据对（的列表）按照单词的词典顺序排列，然后存储到存储器中。数据对是由单词和对应着该单词的倒排列表的引用信息构成的。例如，若用内存上的二叉查找树实现之前例子中的词典，就会得到如图 1-3 所示的树形结构。树中的各个结点是通过地址引用（指针）连接起来的。

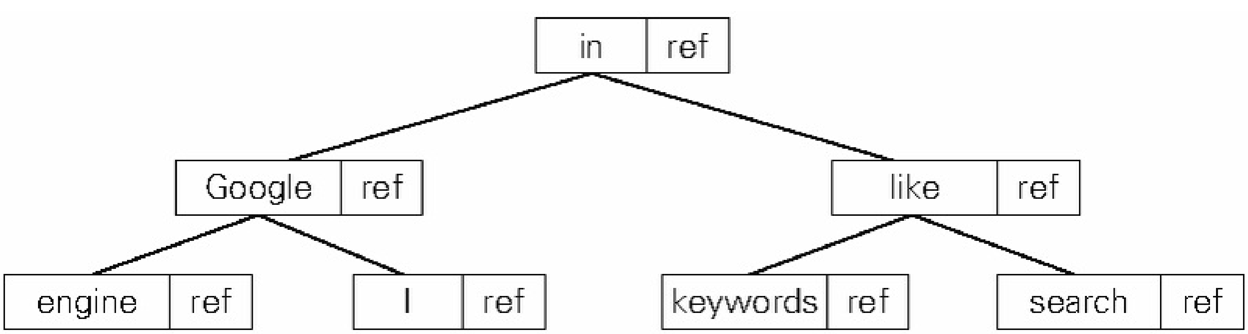


图 **1-3**　在内存上实现词典（使用二叉查找树）

同样地，在二级存储上实现词典时，也要先将数据对按照单词的词典顺序排列，然后一个接一个地存储到存储器上。但是，如果只是单纯地一个接一个地存储，就无法知道各数据对应该在哪里结束了，因此在此之上还要维护一个列表，用于存储从开头算起每个数据对的偏移量。对应的数据结构如图 1-4 所示。在进行检索时，可以对该偏移量的列表进行二分查找。

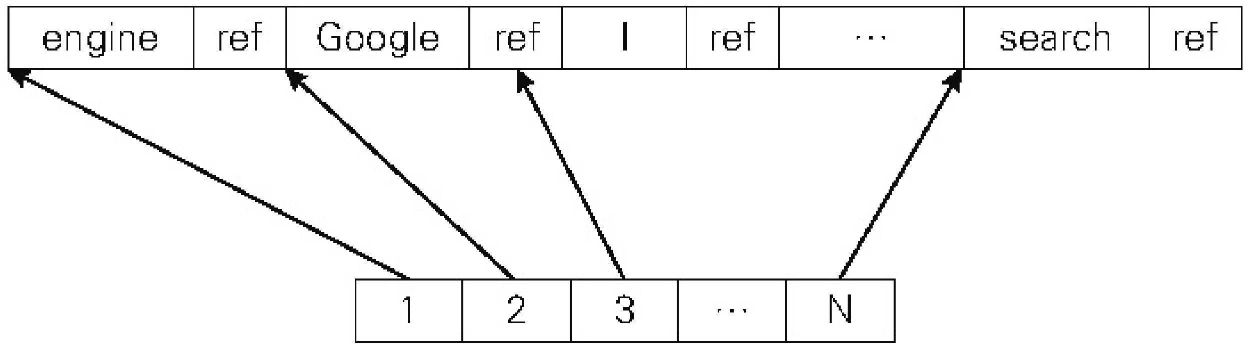


图 **1-4**　在二级存储器上实现词典（使用二叉查找树）

如果词典能够完整地加载到内存，那么所形成的二叉树的搜索效率将会非常高。特别是当二叉树处于平衡状态时，平均进行 $\log_2 N$ 次查找就能找到单词。

但是，如果词典无法完整地加载到内存，而必须存储到二级存储器上时，二叉树就未必是高效的数据结构了。HDD 或 SSD 等二级存储器一般被称作“块设备”，由于它们是以块为单位进行输入输出的 [5]，所以即使只是读取块中 1 个字节的数据，也不得不对整个块进行输入输出操作。例如，假设我们用二叉查找树实现了含有 100 万个单词的词典，那么进行二分查找的话，平均需要 20 次查找，因此在最坏的情况下就需要加载 20 个块。也就是说，假设二级存储的加载性能为 5ms/ 块，那么在 1 次检索中，仅花费在二级存储输入输出上的时间就高达 100ms。

> [5]HDD 的最小输入输出单位是 512 字节的扇区。文件系统通常以页为单位来管理存储空间（空间大小是设备块大小的常数倍），并以页为单位进行输入输出。Linux 通常以 4KB 为一页。

因此，当要存储大型词典时，往往要使用适合块设备的 B+ 树等树形数据结构。

❙ 用 **B+** 树实现词典

B+ 树是一种平衡的多叉树，属于从 B 树派生出来的树形结构。在 B+ 树中，所有的记录都存储在树中的叶结点（Leaf Node）上，内部结点（Internal Node）上只以关键字的顺序存储关键字 [6]。B+ 树的示意图如图 1-5 所示。

[6]由于在数据库管理系统中 B+ 树用得非常普遍，所以也经常可以遇到虽然说的是 B 树，但实际上指的是 B+ 树的情景。

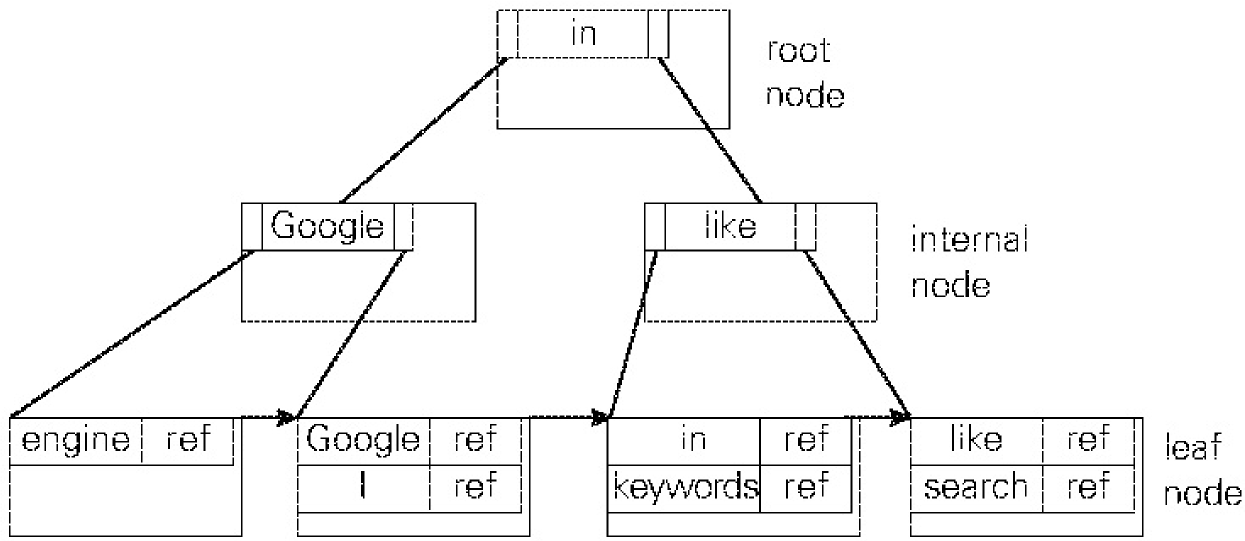


图 **1-5** 在二级存储器上实现词典（使用 **B+** 树）

B+ 树通常以文件系统中页尺寸的常数倍为单位管理各结点，而由这样的结点来构成树，则有助于减少检索时对二级存储的输入输出次数（详细内容请参考书后的参考文献 1）。

下面就让我们用 B+ 树来实现之前的包含了 100 万个单词的词典吧。假设有以下设定。

- 块大小：**4KB**
- 页大小：**4KB**
- 单词的平均大小：**10** 字节
- 页内偏移量的大小：**2** 字节 [7]
- 指向下一级结点的指针的大小：**4** 字节

[7]由于单词的长度不是固定的，所以为了指示出每个单词在页中的保存位置，通常还要维护一个偏移量的数组。

基于这种假设，可以算出每个单词将占用页中 16 个字节的空间，因此

每页中可以存放大约 250 个关键词（单词）[8]。由于页中的每个单词都持有一个指向下级结点的指针，下级结点中存储的是按照词典顺序排在该单词之前（后）的单词集合，所以可以推算出要存储 100 万个单词只需要 3 层结点就足够了（100 万＜ 250× 250×250 ＝约 1500 万）。也就是说，只要从二级存储中读取 3 个结点，就可以检索到任意的单词了。假设二级存储的加载性能还是 5ms/ 块，那么花在检索上的输入输出时间就是 15ms，这与花费在二叉查找树检索上的 100ms 的输入输出时间形成了鲜明的对比。

> [8]为了估算输入输出的次数，这里仅进行了非常粗略地计算。实际上每一页中还包含着用于管理该页信息的头部，而且如果一页中有 N 个单词的话，就还会有 $N+1$ 个指针。

实现倒排文件

在实现倒排文件时，往往会假设所有倒排列表都会变得很长，因此一般都会将倒排列表存储到二级存储的连续区域中 [9]。由此就可以通过二级存储的顺序存取来加载倒排列表了，特别是在使用磁盘驱动器时，这种做法往往能够在数据加载方面增大吞吐量。倒排文件的实现示意图如图 1-6 所示。

> [9]在通过文件系统存储倒排列表时，也可能会遇到文件系统无法为其预留连续存储空间的情况，此时就只能将倒排列表分段存储到多块存储空间上了。等诸位读到了附录部分的动态索引构建时，就能看到我们是如何允许并利用倒排列表的分段存储来努力提升构建索引性能的了。

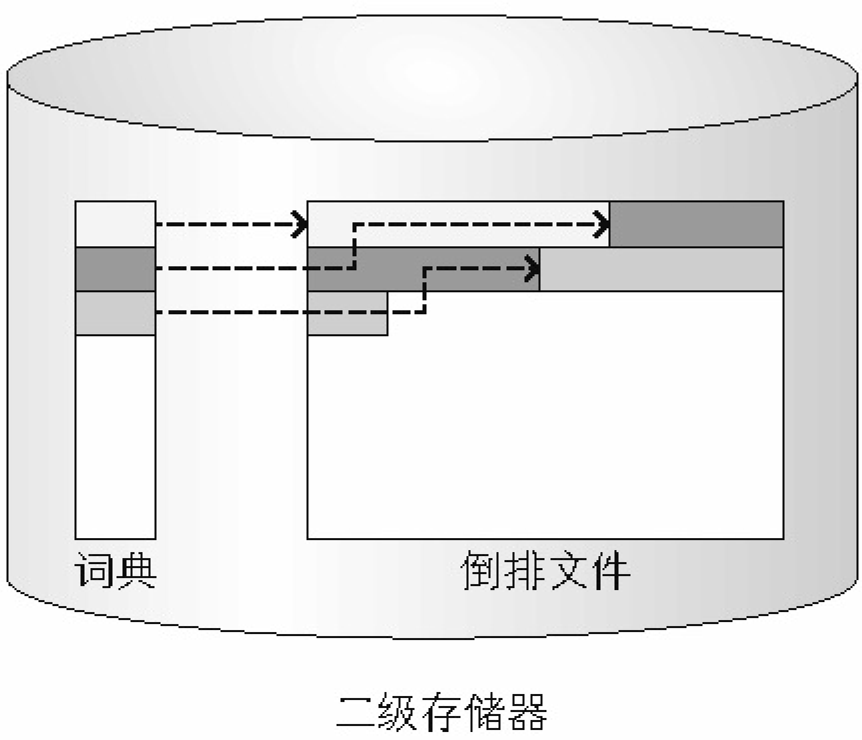


图 **1-6**　实现倒排文件

❚ 倒排列表的物理布局

单词级别的倒排列表由以下两个要素构成。

- 文档编号（**DocID**）
- 文档中的偏移列表（**off1**、**off2...**）

除此以外，各单词在各文档中的出现次数一般也会同时保存。这个次数叫作 TF（Term Frequency，词频），常用于计算检索结果的排名等。

在实现倒排列表时，大多数情况下要将这些数值以二进制形式、按照如下所示的布局存储为二级存储器上的文本（其中的“,”是为了区分各个数据项而额外加上的）。另外，为了进行高效的检索处理，通常还要先将文档编号和偏移量按升序排列后再存储。

DocID,TF,off1,off2,off3

此时，对于之前例子中对应着 search 的倒排列表 (D1;3,D2;2)，就可以

用如下的整数数列表示。

1,1,3,2,1,2

若每个整数都使用 4 个字节表示，那么该整数数列将占用 24 个字节的二级存储空间。另外，在文档级别的倒排列表中，一般会采用只列出文档编号的布局。

▌压缩倒排列表

在检索处理中，由于从二级存储器中读取倒排列表经常会占据大部分的检索处理时间，所以在大多数情况下都会保存经过压缩的倒排列表来缩短加载时间。有关压缩方法我们将在第 5 章详细讲解。由于倒排列表一般都是整数数列，所以通常会采用适合整数数列的压缩方法。

1-6　使用倒排索引进行检索

前面我们已经了解了由索引管理器管理的倒排索引的结构以及具体的实现方法。下面，就让我们再来了解一下在索引检索器上使用倒排索引进行检索的方法吧。

布尔检索

在 1-2 节，我们提到了如何从倒排索引中查找出同时包含多个单词的文档，即先获取与各单词相对应的倒排列表，然后用 AND 运算符计算出其中包含的文档编号的交集。

使用由多个单词通过逻辑运算符连接而成的查询进行检索，称为“布尔检索”（Boolean Retrieval）。逻辑运算符（Boolean Operator）有 AND、OR、NOT 等，其含义分别如下所示。

- **AND**：两边的单词都要包含（逻辑与）
- **OR**：包含任意一边的单词即可（逻辑或）
- **NOT**：不包含某个单词（逻辑非）

另外，我们通常将“如何进行检索”这样的机制称为检索模型，因此执行布尔检索的检索模型就叫作布尔模型。

使用倒排索引的检索处理流程

一般来说，使用倒排索引的检索处理流程如下所示。

① 获取查询中每个单词的倒排列表

② 根据布尔检索，获取符合检索条件的文档编号

③ ' 计算符合检索条件的文档和查询的匹配度

③ '' 获取对检索结果进行排序时使用的属性值

④ 根据匹配度或用于排序的属性值，获取前 k 个文档

另外，虽然在第①步中使用了“每个单词”这一表述，但是要说得严谨一些的话，应该是处理由单词或字符连接而成的每个短语（Phrase）。虽然在此后的讲解中还是使用“单词”这一表述，但是请诸位根据实际情况进行解读。

代码清单 1-1 中列出了上述流程的伪代码。

代码清单 **1-1**　使用倒排索引的检索处理

```
Q = Query // copied
// sort Q by the length of each word's posting list
word ← shift Q
posting_list = fetchList(word) ❶
for all word ∈ Q do
  posting_list2 ← fetchList(word) ❶
  posting_list ← Intersect(posting_list, posting_list2) ❷
end for
array ← newArray() ❸（以下8行）
for all posting ∈ posting_list do
  elem ← newElement()
  elem.val ← calcRelevancy(Query, posting)
  //elem.val ← getAttribute(posting)
  elem.ref ← posting.doc_ref
  push array, elem
end for
// Identify the top-k elem.val and return the corresponding documents. ❹
```

假设在这段伪代码中，我们处理的是由各单词通过 AND 连接而成的布尔查询（单词 1 AND 单词 2 AND … 单词 N）。

首先，根据各自倒排列表长度的升序，对查询中的单词进行排序。之所以这样做是为了在对多个倒排列表两两计算交集的时候，尽可能地减少比较的次数。

接下来，从查询中取出最前面的单词，获取与之对应的（最短的）倒排列表（❶）。

然后，依次计算该倒排列表与查询中剩余单词的倒排列表的交集，最终生成包含全部单词的倒排列表（❷）。接下来，计算刚生成的倒排列表

中的各文档与查询的关联度。此时，若还需要根据日期等属性值而非关联度对检索结果进行排序的话，则还需要从每个文档中获取相应的属性值（❸）。

最后，在按照关联度和属性值对检索结果进行排序后，取出检索结果中的前 k 个文档（❹）[10]。在取出结果的时候，能将所有结果都提供给用户固然好，然而当结果数量过多时，通常只提供根据某种标准选出的“优质结果”。

> [10]若无需对整个检索结果排序，而是仅对前 k 条结果排序即可的话，可以利用堆结构进行高速排序。

另外，若查询是由短语构成的，则还需要在第❶步进行查找短语的处理，具体的方法我们在 1-3 节中介绍过。由于篇幅有限，此处仅仅列出了有关 AND 查询的伪代码，对于其他的布尔操作符，也可以用与此大致相同的流程进行处理。

关联度的计算方法

在 Web 搜索引擎中，一般是按照文档与查询的关联度对检索结果进行排序的。计算关联度的方法有余弦相似度（Cosine Similarity）和 Okapi BM25 等。

在计算余弦相似度时，需要把文档和查询映射到以单词（Term）为维度的向量空间上，文档向量和查询向量的夹角（内积）越小，说明文档和查询的关联度越高。

而 Okapi BM25 则是基于“文档是否匹配查询是由概率决定的”这一统计原理，根据单词的出现频率等因素计算出查询与文档相关联的概率，这个概率越大，说明文档和查询的关联度越高。

关于这些方法就先简单介绍到这里，有兴趣的读者可以参照书后的参考文献 2、3。

信息检索中的检索

在被称为信息检索的全文搜索学术领域中，由于其原本的目的就是找出与信息需求相匹配的文档，因此可以认为匹配的文档中没有必要包含查

询。也就是说，在检索处理中，文档是否包含查询无关紧要，重要的是通过计算查询和整个文档的关联度，把关联度高的文档作为检索结果。

代码清单 1-2 中列出了将余弦相似度作为关联度的指标来进行检索 处理的伪代码。

代码清单 **1-2**　检索时只考量了文档和查询的关联度

```
// Calculate word vector Vq,w for each word w in Query
for all d ∈ Documents do
  Sd ← 0
  for all word ∈ Query do
    Calculate document vector Vd,w
    Sd ← Sd ＋ Vq,w · Vd,w // calculate the inner product of the two
vectors
  end for
  Wd ← getDocumentLength(d)
  Sd ← Sd/Wd // normalize the score
end for
// Identify the k greatest Sd values and return the corresponding documents
```

在该方法中，因为要计算的是所有文档和查询的关联度，所以作为检索对象的文档越多，检索处理的成本就越高。

与此相对，若是先将检索对象限定为至少包含 1 个查询中的单词的文档，再计算关联度的话，就可以降低检索处理的成本了。这种情况下的检索处理伪代码如代码清单 1-3 所示。

代码清单 **1-3**　检索时考量了至少包含查询中 **1** 个单词的文档和查询的关联度

```
// Allocate an accumulator Ad for each document d and set Ad ← 0
for all word ∈ Query do
  // Calculate word vector Vq,w
  posting_list ← fetchList(word)
  for all (d, freq) ∈ posting_list do
    // Calculate document vector Vd,w
    Ad ← Ad ＋ Vq,w · Vd,w // calculate the inner product of the two
vectors
  end for
end for
```

```
Wd ← getDocumentLength(d)
Ad ← Ad/Wd for each Ad // normalized the score
// Identify the k greatest Ad values and return the corresponding documents
```

由此可见，在搜索引擎中各种各样的方法都可用于检索处理。而搜索引擎的开发者则需要根据作为检索对象的文档的性质和检索应用程序的用途，适当地选择这些方法。

1-7　构建倒排索引

前面我们已经了解了由索引管理器管理的倒排索引的结构以及在索引检索器上进行检索处理的流程。下面，就让我们再来看一下如何在索引构建器上构建倒排索引吧。

使用内存构建倒排索引

生成了与文档编号对应的单词表后对该表进行倒排，在 1-2 节我们通过这种方法生成了倒排索引。若让计算机来处理的话也是如此，先在内存上生成与文档编号对应的单词表（二维数组），然后用相同的方法倒排该表，就可以构建出倒排索引了。但是，由于大多数情况下倒排索引都是非常稀疏的表，因此这种构建方法可能会消耗大量的内存。

于是就有了用链表实现倒排列表这一优化方法。相比之下，该方法只需少量的内存就可以构建出倒排索引。

使用二级存储构建倒排索引

在今天的硬件环境下不乏装配有大量内存的计算机，尽管如此，在很多情况下还是需要处理超过实际内存量的大规模文档。遇到这种情况时，可以利用二级存储来构建索引。作为利用二级存储构建索引的代表性方法，本书将会介绍“基于排序的构建方法”和“基于合并的构建方法”。为了使文章简明扼要，下面只介绍如何构建文档级别的倒排文件，但是其中步骤也适用于构建单词级别的倒排文件。

❙ 基于排序的索引构建法

基于排序的索引构建法是一种将由单词和倒排项组成的二元组写入

二级存储，并以单词的词典顺序对这些二元组排序，以此来构建倒排索引中的倒排列表的方法。具体的构建程序如代码清单 1-4 的伪代码所示。

首先，对各文档中构成该文档的每个单词都建立一条形如“单词、文档编号、单词在文档中的出现次数（TF）”的记录，然后将该记录写入到

二级存储上的文件的末尾（❶）。

接下来，将文件中的各条记录优先按照单词的升序排列，单词字段相同的记录需再按照文档编号的升序排列（❷）[11]。

> [11]这时的排序方法通常会选择合并排序，例如像下面这样做：首先以块的整数倍为单位将多条记录加载到内存中，然后对这些记录进行快速排序并将排序后的记录导出到文件中，最后利用多路合并排序将多个导出的文件合并在一起。

最后，从第一行开始逐行地读取排序后的文件，取出每个单词的文档编号的列表，并用这些列表构建出各个单词的倒排列表（❸）。此外，压缩倒排列表的操作也是在❸的步骤中进行。

代码清单 **1-4**　基于排序的索引构建法

```
file ← newFile()
while all documents have not been processed do
  d ← getDocument()
  for all word∈ d do ❶（以下3行）
    appendToFile(file, word, d.docID, count(d, word))
  end for
end while
externalSort(file) ❷
inverted_file ← newFile()
for all r ∈ file do ❸（以下4行）
  posting ← constructPostingList(r.docID, r.freq)
  add(inverted_file, r.word, posting)
end for
```

图 1-7 是基于排序的索引构建处理的示意图。

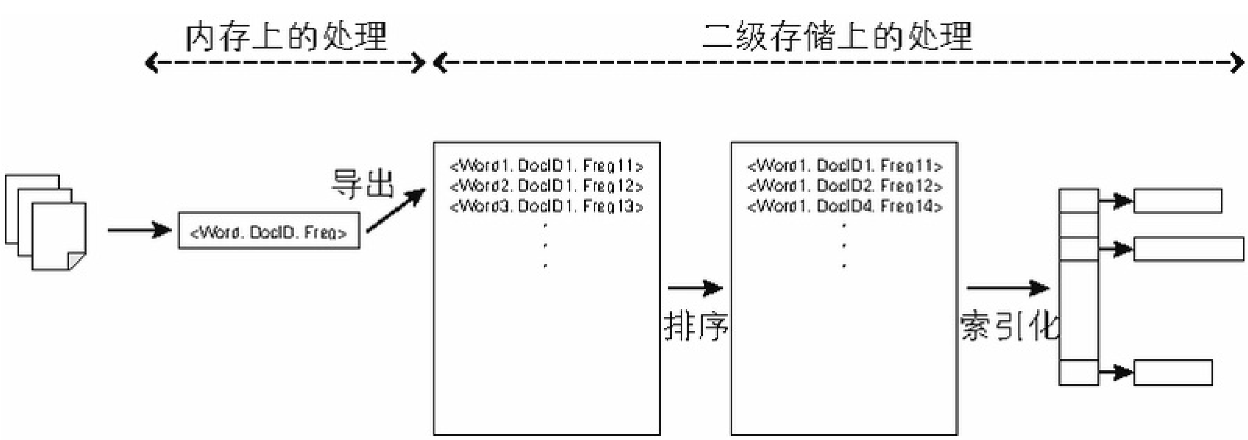


图 **1-7**　基于排序的索引构建法

■ 基于合并的索引构建法

基于合并的索引构建法是一种先在内存上构建出倒排索引的片段，然后将这些片段导出到二级存储，最后将导出的多个倒排索引片段合并在一起，以此来构建最终的倒排索引的方法。具体的构建程序如代码清单 1-5 的伪代码所示。

代码清单 **1-5**　基于合并的索引构建法

```
n ← 0
while all documents have not been processed do
  n ← n ＋ 1
  filen ← newFile()
  map ← newMap()
  while free memory available do ❶（以下11行）
    d ← getDocument()
    for all word ∈ d do
      if word ∈ map then ❷（以下2行）
        posting_list ← newPostingList()
      else
        posting_list ← getPostingList(map, word)
      end if
    add(posting_list, d.docID)
    end for
  end while
  sorted_map ← sort(map) ❸（以下2行）
  writeToFile(filen, sorted_map)
end while
filemerged ← mergeFiles(file1, ... , filen) ❹
```

首先，在内存上构建出以单词为键，以倒排列表为值的映射表（Mapping），即由部分数据构成的倒排索引的片段（❶）。

每当遇到文档中的单词不在映射表中时，都要将该单词加入到映射表中（❷）。

当映射表的大小（事先已设定好）达到内存大小的上限时，就将该映射表导出到文件中（❸）[12]。

[12]在这段伪代码中，由于是将哈希表用作词典，所以要在导出时进行排序。然而若使用像树那样的、带有顺序的数据结构来管理词典的话，就可以省略排序的步骤了。

像这样反复处理，直到处理完所有的文档，最后利用多路合并将导出的多个文件合并在一起，构建出最终的倒排索引（❹）。另外，压缩倒排列表的操作也是在❹的步骤中进行。

基于归并的索引构建处理的示意图如图 1-8 所示。

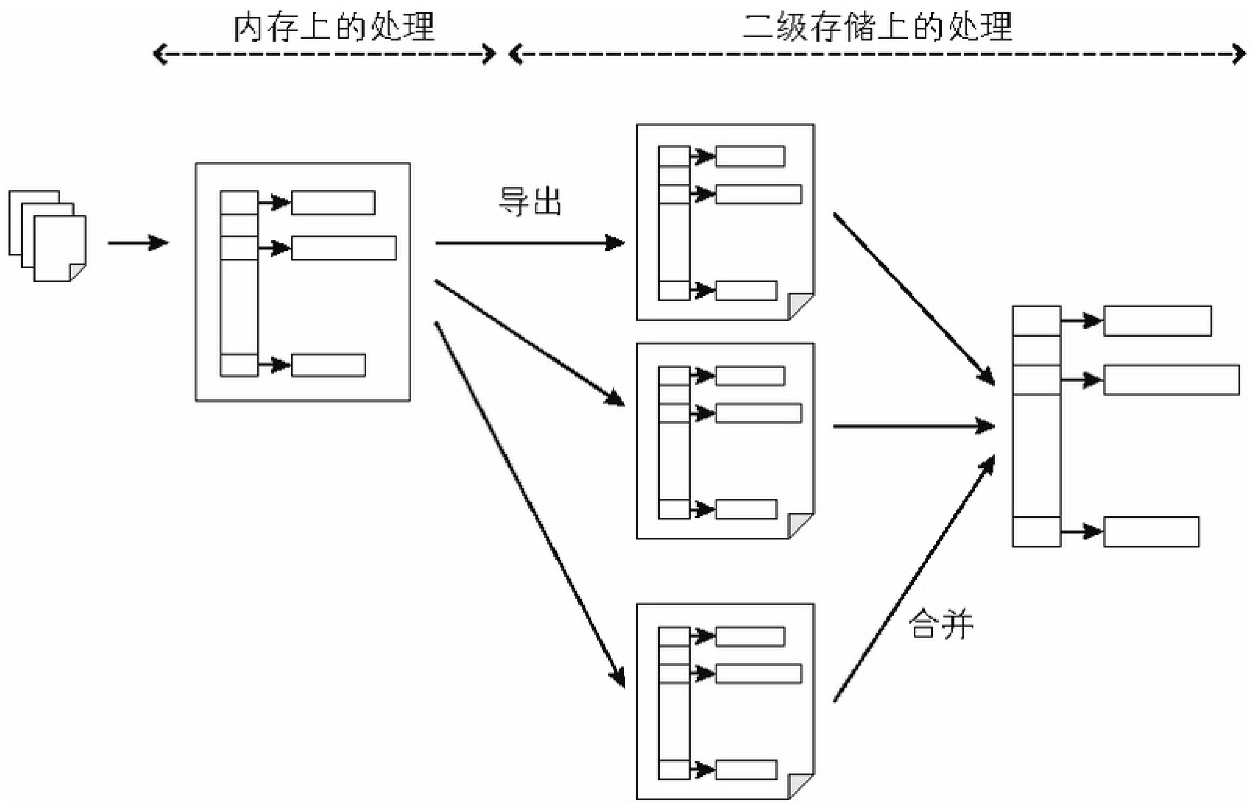


图 **1-8**　基于合并的索引构建法

虽然本节省略了对相关算法的详细分析，但是一般认为基于合并的索引构建法拥有更高的效率。这是由于相对于基于合并的方法，基于排序的方法要在二级存储上进行排序，所以读写的总量往往会增多。

静态索引构建和动态索引构建

之前讲解过的索引构建方法都是对输入数据进行批量构建处理。也就是说，在构建处理完成之后索引才能用于检索。在信息检索领域中，这样

的构建方法被称为“静态构建方法”（Offline Index Construction）。静态构建方法多用于文档集合较稳定，或是即使文档集合发生了变化，在变化同步到索引之前还有一定时间等场景。

与此相对，还有一种“动态构建方法”（Online Index Construction/Dynamic Indexing）。这种方法不但可以使索引结构时刻处于可供检索的状态，还可以一边实时更新索引，一边构建索引。这种方法多用于信息的时效性非常重要的文档，例如 Web 上的新闻或博客中的文章等。在书后的附录部分中我们将会详细地讲解动态构建方法。

1-8　准备要检索的文档

前面我们以“想要进行检索的数据已经在手边了”为前提，讲解了搜索引擎的结构及其构成要素。但是，在实际中这些数据来自哪里呢？在本章的最后，让我们再来大略地了解一下如何收集、整理作为检索对象的数据吧。

收集数据

搜索引擎中作为检索对象的数据来自哪里呢？

一种情况是要检索的数据已经存在了。有时是大量的数据已经存在于企业的文件服务器、邮件服务器，或每个人的 PC 中了；有时是运营博客或社会化书签等 Web 应用程序的公司，已将由应用程序的用户添加、更新的信息存入数据库等系统中了。在这种情况下，数据并不是我们亲自收集的，而是自然而然地储存起来的。

与此相对，还有一种是难以亲自收集数据的情况。例如，在 Web 检索中，要借助前文所述的称为爬虫的软件遍历 Web，才能将全世界的 Web 网页收集起来。而当我们想在 Twitter 外使用 Twitter 的检索服务来收集推文时，还要利用 Twitter 的 API 等。

要收集的数据越多，收集、存储这些数据的机制就越复杂。例如，在需要保存大量数据的情况下，高效地管理存储器也是必不可少的环节。为了通过 Web 应用程序保存大量的访问记录，应用程序的高度可扩展性就显得尤为重要。而且，为了使爬虫能够高效地运转起来，还必须设法优化遍历 Web 的算法。

由于这些都不是针对搜索引擎的技术，而且也超过了本书的讨论范围，所以我们就简单介绍到这里。但是如果要开发、运维能够处理大规模数据的搜索引擎，还是不可避免地要去攻克诸如此类的、各种各样的技术难题。

数据规范化

以某种方法收集而来的数据只有经过处理才能成为适合搜索引擎检索的

文档。例如、由爬虫收集而来的 HTML 文件除了包含内容还包含了标签等标识页面结构的信息；而文件服务器中的 PDF 文件存储了 由私有的二进制结构表示的内容。这些文件都包含着不利于检索的信息，因此要将它们规范为只包含要查询的文字信息或文章内容的文档。例如，在规范 HTML 文件时，就要删除标签并提取出作为检索对象的文章（内容）。

另外，提取出来的文章也未必都是按照规则书写的。例如，可能会遇到在某篇文章中使用的是全角的数字和字母，而在另一篇文章中使用的又是半角的数字和字母这种情况。在这种情况下，就要按照某种约定好的规则（例如统一使用半角字符）来使文档规范化。相应地，只要对查询也进行了同样的规范化，就能查找到（匹配到）规范化之前无法找到（无法匹配）的文档。

因此，大多数提供检索服务的系统都会先转换收集而来的数据，使其格式适合作为搜索引擎的输入文档。

至此为止，有关搜索引擎基本概念的讲解就结束了。诸位辛苦了。倒排索引是一种非常简单的结构，因此理解起来应该并不吃力。如果仅仅是“想制作一个检索系统”，那么以目前为止讲解过的基础知识为基础，并借助开源的搜索引擎 [13]，应该就可以在较短的时间内将其实现了。尽管如此，但如果一味地利用已有的搜索引擎，可能就很难深入地了解搜索引擎的原理，或是优化现有的搜索引擎了。那么应该怎么做呢？笔者认为，动手制作一个简单的搜索引擎正是实现这些目标（能够制作出来、深入了解原理、能够进行优化）的最好方法。基于这种想法，从第 2 章开始，就让我们一边通过源代码从内部梳理搜索引擎的工作流程，一边体验搜索引擎的开发过程吧。请诸位一定要一边实际动手做，一边更加深入地去探索搜索引擎的结构。

> [13]从使用了 Apache Lucene 或 Lucene 的 Solr 和 Elasticsearch，到笔者（未永）也参与了开发的 Groonga 和 Senna 等，都是很优秀的软件。

第 **2** 章　准备全文搜索引擎的检索样本

在本书中，为了理解搜索引擎的核心，我们开发了一个叫作 wiser 的、仅实现了最基本功能的全文搜索引擎。诸位可以从下面的地址下载 wiser 的源代码。

http://www.ituring.com.cn/book/1582（点击**“**随书下载**”**）

在第 2 章中，我们会先大致介绍一下 wiser 的概要，然后开始着手搭建 wiser 的运行环境。

2-1　全文搜索引擎 wiser

wiser 的构成

wiser 的系统构成及其源代码的目录结构如图 2-1 所示。下面，请诸位一边回忆在第 1 章的开头部分讲解过的搜索引擎的构成，一边来看下面这张图。

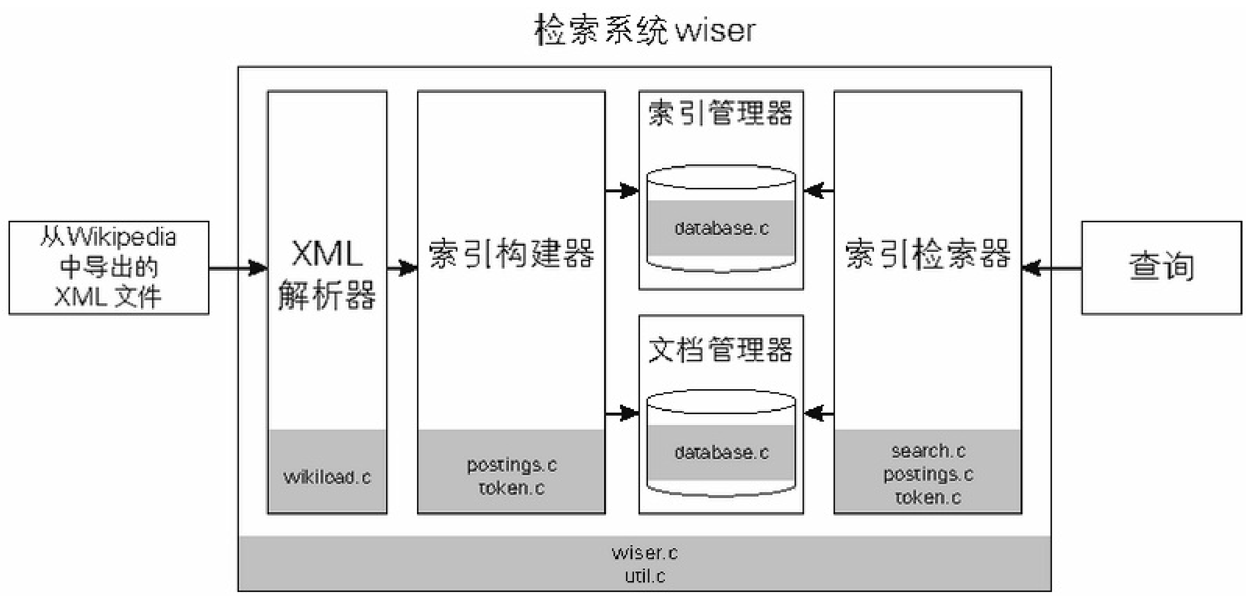


图 **2-1**　全文搜索引擎 **wiser** 的构成

wiser 中所有的处理过程都是从 wiser.c 开始的。wiser.c 会先去解析命令行的参数，然后根据参数调用构建索引或执行检索的处理过程。

“XML 解析器”是由 wikiload.c 实现的，负责从 Wikipedia 的副本中提取文本数据形成文档，并将文档储存到数据库中。

“索引构建器”是由 postings.c 和 token.c 实现的，负责将文本文档转换为索引。

“索引检索器”是由 search.c、postings.c 和 token.c 共同实现的，负责使用倒排索引进行检索处理。

database.c 负责借助名为 SQLite[1] 的 RDBMS（关系型数据库管理系统）

管理文档数据和索引。

> [1]http://www.sqlite.org/

util.c 负责提供在所有模块中都会用到的通用处理。

准备用于检索的文档

正如第 1 章所述，收集大量的数据是件辛苦的工作，因此在本书中，我们将使用中文版的 Wikipedia 作为检索对象。诸位可以从以下地址下载包含 Wikipedia 中所有词条的压缩文件。

http://dumps.wikimedia.org/zhwiki/

截至 2015 年 7 月，该压缩文件的大小约为 1.1GB，解压缩后的文件大小为 4.9GB。要检索的数据量对于 grep 等工具来说确实较大，但是又没有大到 1 台机器无法处理的程度，因此这个量级的数据对于 wiser 来说正合适。

Wikipedia 的词条是由如下所示的 1 个巨大的 XML 文件构成的。

```
<mediawiki>
  <page>
    <title>词条的标题</title>
    <revision>
      <text><![CDATA[
词条的正文
]]>
      </text>
    </revision>
  </page>
  <page>
    …
  </page>
  …
</mediawiki>
```

可以看到，每一个词条都包含在一对 page 标签中。title 标签中的内容是词条的标题，位于 revision 标签中的 text 标签中的内容是词条的正文。在本书中，我们会提取各个词条的标题和正文，并将这两部分数据视为

要检索的文档。负责从 XML 文件提取词条的 wikiload.c 会进行如下的处理。

- 根据指定的路径打开 **XML** 文件
- 获取 **<title>** 标签中的内容作为文档的标题
- 获取 **<text>** 标签中的内容作为文档的正文
- 将标题和正文传给索引构建器反复执行上述处理过程中的后 **3** 步**……**

在本章中，我们先不讲解上述处理的细节。具体的处理过程请参阅附录 A-2。

2-2　安装 wiser

构建 wiser

下面我们讲解在 Linux 发行版 CentOS 和 Debian（Ubuntu），以及 Mac OS X 上安装 wiser 的方法。为了构建 wiser，我们需要先安装 sqlite 和 expat[2] 的代码库。另外，为了解压缩 Wikipedia 的副本，我们还需要先安装 bzip2。

> [2]http://expat.sourceforge.net/

在 CentOS 上安装 wiser

在 CentOS 上，我们可以使用包管理工具 yum 来安装构建环境及 wiser 所依赖的代码库。

示例

```
> yum install gcc sqlite sqlite-devel expat-devel bzip2
```

在 Debian 上安装 wiser

在 Debian 上，我们可以使用包管理工具 aptitude 来安装构建环境及 wiser 所依赖的代码库。

示例

```
> aptitude install build-essential sqlite3 libsqlite3-dev libexpat1-dev bzi
```

在 Mac OS X 上安装 wiser

在 Mac OS X 上安装 wiser 时，我们需要先安装 Xcode[3] 作为构建 环境。

> [3]https://developer.apple.com/xcode/

然后，使用称为 Homebrew[4] 的包管理工具来安装 wiser 所依赖的代码库。如果事先没有安装 bzip2，接下来还需要另行安装。

> [4]http://mxcl.github.io/homebrew/

示例

```
> brew install sqlite expat
```

依赖的代码库都装好后，就可以进入到 wiser 的源代码目录了。此时，只需要执行 make 命令，即可在当前目录中生成一个名为 wiser 的可执行文件。

示例

```
> make
```

启动 **wiser**

下面就让我们启动 wiser 看看吧。

先不带任何参数地执行一次，这样做的目的是确认 wiser 的使用方法。

示例

```
> wiser
usage: ./wiser [options] db_file

options:
  -c compress_method             : compress method for postings list
  -x wikipedia_dump_xml          : wikipedia dump xml path for indexing
  -q search_query                : query for search
  -m max_index_count             : max count for indexing document
  -t ii_buffer_update_threshold : inverted index buffer merge threshold
  -s                             : don't use tokens' positions for search

compress_methods:
none   : don't compress.
golomb : Golomb-Rice coding(default).
```

可以看到 wiser 支持如表 2-1 所示的参数。

表 **2-1　wiser** 的参数

参数	作用
-c	设定构建倒排索引时使用的压缩方法
-x	设定导入 Wikipedia 副本的路径
-q	设定检索时使用的查询字符串
-m	设定最多从副本中导入多少个文档
-t	设定更新倒排索引时使用的缓冲区大小
-s	进行检索时不使用词元的位置信息，即不进行短语检索

解压缩 **Wikipedia** 的副本

诸位可以从下面的 URL 下载中文版 Wikipedia 的最新副本。

http://dumps.wikimedia.org/zhwiki/latest/zhwiki-latest-pages-articles.xml.bz2

然后，可以通过 bunzip2 命令将该副本解压缩。

示例

```
> bunzip2 -k zhwiki-latest-pages-articles.xml.bz2
```

2-3　运行 wiser

构建倒排索引

既然 wiser 的准备工作已经就绪了，那么接下来就让我们导入 Wikipedia 的词条[5]，开始构建倒排索引吧。只需运行如下的命令，即可构建出倒排索引。

> [5]Wikipedia 的副本中包含繁体中文的词条。——译者注

示例

```
> ./wiser -x zhwiki-latest-pages-articles.xml -m 1000 wikipedia_1000.db

[time] 2015/11/04 02:35:31.000005
count:1 title: Wikipedia:Upload log
count:2 title: Wikipedia:删除记录/档案馆/2004年3月
count:3 title: 数学
count:4 title: Help:目录
count:5 title: 哲学
count:6 title: 文学
```

在上面的例子中，我们通过将参数 -m 的值设定为 1000，使 wiser 最多就加载 1000 个文档。之所以暂时控制在 1000 个文档以内，是因为这里我们只想确认 wiser 能否正常运行。

另外，在运行 wiser 时，还需要设定倒排索引文件的文件名。在本例中使用的文件名是 wikipedia_1000.db，不过诸位也可以随意命名该文件。

使用倒排索引查询

下面就让我们使用倒排索引检索一下“语言”这个单词吧。

示例

```
> ./wiser -q "语言" wikipedia_1000.db
[time] 2015/07/17 07:28:44.000009
```

```
document_id: 64 title: Help:跨语言链接 score: 99.842498
document_id: 83 title: 语言列表 score: 64.532346
......
document_id: 88 title: Help:搜索 score: 1.217591
document_id: 96 title: 中国 score: 1.217591
Total 43 documents are found!
[time] 2015/07/17 07:28:45.000000 (diff   0.031247)
```

诸位都顺利地检索出结果了吗？

索引很强大吧，借助索引，检索一瞬间就返回了结果。

从命令行可以看出，要想进行检索，就需要先在参数 -q 的后面设定要查询的单词，然后再在后面接上构建索引时设定的索引文件的文件名。

掌握了检索方法后，可以试着再检索一些其他的单词。

比较 **grep** 和 **wiser** 的运行速度

通过建立倒排索引，到底能使检索时间缩短到什么程度呢？也许与 grep 比较一下就能知道答案了。为了能在相同的条件下进行比较，我们需要先制作一个只包含 1000 个文档的 XML 文件。

示例

```
$ grep -m 1000 -n '</page>' zhwiki-latest-pages-articles.xml | tail -n 1 |
cut -d ":" -f 1 | xargs -I LINE head -n LINE zhwiki-latest-pages-articles.x
 > 1000.xml
```

上述命令的含义是先用 grep 命令从 XML 文件中筛选出含有“</page>”的前 1000 行，然后通过 tail 和 cut 命令获取最后一行的行号，接着将整个 XML 文件中从第 1 行到这一行的数据导出到指定文件中。

接下来就用这个 XML 文件来比较一下 grep 和 wiser 的检索速度吧。

示例 [6]

[6]如果当前使用的是默认的 Bash Shell，并且想得到示例中的输出格式，那么可以执行如下命令 /usr/bin/time--format="%C %Us user %Ss system %P cpu %U total" grep 'Wikipedia' 1000.xml，即

> 需要执行位于“/usr/bin/”下的 time 命令，并设定 format 参数。有关百分号后字母的含义，可参阅 time 命令的手册。——译者注

```
> time grep 'Wikipedia' 1000.xml
...
grep 'Wikipedia' 1000.xml 0.50s user 0.01s system 76% cpu 0.664 total

> time ./wiser -q 'Wikipedia' wikipedia_1000.db
...
./wiser -q 'Wikipedia' wikipedia_1000.db 0.01s user 0.00s system 83% cpu
0.021 total
```

wiser 的检索速度真是太快了。相对于 grep 花费的 0.50 秒，wiser 仅仅用了 0.01 秒就返回了检索结果。而且，要检索的文档越多，二者在检索速度上的差距就越明显。

第 **3** 章　构建倒排索引

我们在第 2 章确认了 wiser 的构成和运行结果。现在进入第 3 章，终于可以接触到搜索引擎的核心部分了，即倒排索引的实现过程以及构建倒排索引的过程。相关的基础知识已经在第 1 章讲解过了，因此在阅读本章时若有疑问，不妨回过头去读读第 1 章。另外，在本章我们并没有对构建出来的倒排索引进行压缩处理。有关压缩处理的讲解和实现将在第 5 章中进行。

3-1 复习有关倒排索引的知识

提取词元

既然 wiser 采用的索引格式是倒排索引，我们就先来简单地复习一下构建倒排索引的步骤吧。

- 从作为检索对象的文档中提取出词元及其出现的位置
- 对于每个词元，将其所在文档的引用信息（文档编号）和出现在文档中的位置保存起来

以上就是构建倒排索引的过程。在这个过程中，我们还需要利用 N-gram 或词素解析的方法将句子分割成词元的序列。

从句子中分割出词元的处理看似简单，但是在处理中文时，还是要稍加注意才行。之所以这样说是因为 Wikipedia 的词条都是用 UTF-8 的字符编码表示的，因此在进行处理时，不得不考虑这种字符编码的特性。

在 UTF-8 中，是用 1 到 4 个字节的长度来表示 1 个字符的。例如，像数字和拉丁字母等在英文中使用的字符都是用 1 个字节表示的，而在中文中使用的字符则多半要用 3 个字节才能表示。因此，在 UTF-8 中，“Web 检索”这个字符串就是由如下所示的含有 9 个字节的字节序列表示的。

```
0x57 0x65 0x62 0xe6 0xa3 0x80 0xe7 0xb4 0xa2
```

字节序列和各个字符的对应关系如图 3-1 所示。

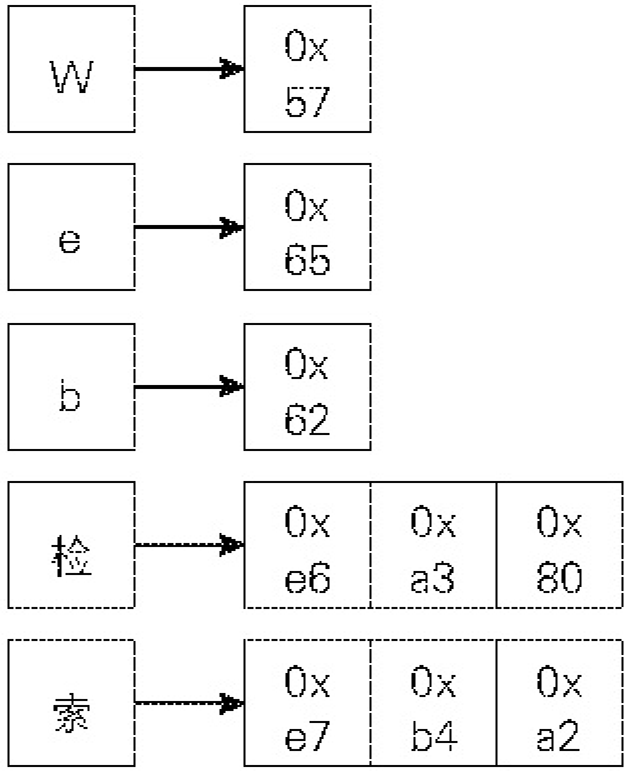


图 **3-1**　在 **UTF-8** 中，字符串中各个字符对应的字节序列

在 wiser 中，由于我们采用了 $N = 2$ 的 N-gram（bi-gram）将句子分割为词元的序列，所以每个词元中都包含 2 个字符。但是在分割过程中，由于分割出来的词元所含有的字节数并不固定，所以还必须分别考 虑每个词元的分割位置。例如，由于刚刚的“Web 检索”中既有英文又有中文，所以提取出的词元会分别出现在该字符串中第 0、1、2、3、6 字节的位置上。由于这几个数之间并没有固定的间隔，所以不能简单地以 3 个字节为单位分割句子。

在 wiser 中，为了避开由使用 UTF-8 带来的处理上的麻烦，我们在每次获取 N-gram 时，都会先将字符串的编码从 UTF-8 转换成 UTF-32。UTF-32 是一种以 4 字节（32 bit）的数值为单位表示 Unicode 字符的编码方式。由于 Unicode 的字符与表示该字符的数值是一一对应的，所以在 UTF-32 中，由 N-gram 分割而成的词元所含有的字节数就变成固定的了，这样就简化了程序上的处理过程 [1]。可是从处理速度和内存使用量的角度来看，一般认为原封不动地使用 UTF-8 处理的效果更好，但是为了让示例程序更易于理解，还是请允许我们在讲解时使用 UTF-32 吧。

> [1]Unicode 中含有一种称作组合字符（Combining Character）的特殊字符。组合字符包括可附着于其他字符之上的符号等，作用是与已有的字符组合起来形成 1 个新的字符。例如，在 Unicode 中，就为诸如汉语拼音的“ü”上的两个点（分音符 U+0308）等符号分配了单独的编码。这就造成了同一个字符“ü”对应着“0xc3 0xbc”和“0x00 0x75、0xcc 0x88”两个编码。因此一旦出现了组合字符，UTF-32 下的 1 个字符就不再只对应于 1 个编码了。但是在本书中，我们还是会将 UTF-32 下的 1 个编码视作 1 个字符处理。也就是说，忽略组合字符的存在。

在 wiser 中，由文件 token.c 中的函数 ngram_next() 负责将句子分割成词元。

为每个词元创建倒排列表

将句子分割成词元以后，要做的就是为每个词元创建倒排列表。正如第 1 章所述，倒排列表要么是关联到词元上的文档编号的集合，要么是由文档编号和词元在文档中出现的位置构成的二元组的集合。我们称前者为文档级别的倒排列表，后者为单词级别的倒排列表。另外，将由所有词元的倒排列表汇聚而成的集合称为倒排文件。

在 wiser 中，我们采用的是单词级别的倒排列表。因此，正如第 1 章所述，这意味着可以利用 wiser 进行快速的短语检索。

下面就让我们看看在 wiser 中是如何构建倒排索引的吧。

3-2　构建倒排索引

在存储器上创建倒排列表

在本书中，由于用作样本数据的 Wikipedia 的文档相对较大，所以只在内存上为所有文档构建倒排索引并不现实。因此，用 wiser 构建倒排索引时，我们还要使用硬盘或 SSD 等存储器（二级存储器）。

在 wiser 中，我们将采用一种新方法构建倒排索引，该方法源自在第 1 章中讲解过的基于归并的构建方法。大致描述一下这个方法的话，就是对于某个文档集合，先在内存上为其建立一个较小的倒排索引，然后将这个较小的倒排索引和存储器上的倒排索引合并，通过反复进行这两步操作，最终就能一点点地在存储器上构建出较大的倒排索引了。

其实如果想要在存储器上创建倒排列表，最直接的方法就是不断地将倒排项（文档编号和位置信息）添加到存储器上的倒排列表的末尾。但是，为了简化将在第 5 章中讲解的压缩倒排列表的过程，我们最终还是采用了上述方法。

下面我们就来看一下倒排索引的数据结构与构建方法吧。另外，在本书中，我们将在内存上构建的临时倒排索引称为“小倒排索引”。

倒排列表和倒排文件的数据结构

在 wiser 中，倒排列表是使用结构体 postings_list 来管理的。该结构体的结构如下所示，各元素的用途请参考变量名后面的注释。

```
/* 倒排列表（以文档编号和位置信息为元素的链表结构） */
typedef struct _postings_list {
  int document_id;             /* 文档编号 */
  UT_array *positions;         /* 位置信息的数组 */
  int positions_count;         /* 位置信息的条数 */
  struct _postings_list *next; /* 指向下一个倒排列表的指针 */
} postings_list;
```

另外，在 wiser 中，我们还使用了名为 utarray[2] 的代码库来处理数组。

在 utarray 中，数组是用 UT_array 类型的指针来表示的。

> [2]http://troydhanson.github.io/uthash/utarray.html

在 wiser 中，倒排文件是使用另一个结构体 inverted_index_hash 来管理的。该结构体的结构如下所示。乍一看 inverted_index_hash 类型不过是个普通的结构体，但它实际上是个关联数组，管理着关联到词元编号上的倒排列表。该关联数组的结构将在稍后讲解。

```
/* 倒排索引（以词元编号为键，以倒排列表为值的关联数组） */
typedef struct {
  int token_id;                  /* 词元编号（Token ID）*/
  postings_list *postings_list; /* 指向包含该词元的倒排列表的指针 */
  int docs_count;                /* 出现过该词元的文档数 */
  int positions_count;           /* 在所有文档中该词元的出现次数之和 */
  UT_hash_handle hh;             /* 用于管理倒排列表的关联数组 */
} inverted_index_hash, inverted_index_value;
```

虽然一边遍历倒排列表的链表，一边计数也能统计出出现过某个

词元的文档数（docs_count），但是若每次检索时都要统计一遍的话就会影响效率。因此，在构建索引的阶段，我们要先将统计好的文档数存储起来。同理，还要将词元的出现次数（positions_count）也一并存储起来。

在结构体 inverted_index_hash 中还存在着一个类型为 UT_hash_handle 的变量 hh，用于将结构体当作关联数组来处理。在 wiser 中，为了处理关联数组，会使用名为 uthash[3] 的代码库，将一个像 hh 那样的类型为 UT_hash_handle 的成员变量添加到结构体的定义中。仅需如此，我们就可以像处理关联数组一样处理结构体了。

> [3]http://troydhanson.github.io/uthash/userguide.html

不过可能有这样一点不太好理解。在一般情况下，关联数组本身的类型和存放在关联数组中的值的类型是不相同的。但是，在 uthash 中，二者的类型却是一样的。二者使用了相同的类型会导致仅凭类型名称判断不出实际上要处理的对象是什么，因为一个关联数组的类型既可以表示要处理的是整个关联数组，也可以表示要处理的仅仅是关联数组中的一个

元素。于是，我们通过为类型赋予别名使二者有所区别，用 inverted_index_hash 类型表示整个关联数组，用该类型的别名 inverted_index_value 表示关联数组中的一个元素。虽然有些复杂，但是这些都是为了使用 uthash 处理关联数组而必须遵守的约定，因此在这一点上还望诸位能够理解。

另外，虽然在第 1 章就讲过“倒排索引是由词典和倒排文件构成的”，但是在实现 wiser 时，我们并没有严格地区分二者。在 wiser 中，将词元及其编号关联起来的数据结构充当了所谓的词典，inverted_index_hash 类型的数据结构充当了倒排文件。

从源代码级别梳理倒排索引的构建顺序

下面，我们将对照着源代码讲解有关倒排索引的构建顺序。将在本章讲解的函数及定义了该函数的文件如表 3-1 所示。

表 **3-1**　函数名和定义了该函数的文件

函数名	文件名	是否会在本章详细讲解
add_document()	wiser.c	○
create_new_inverted_index()	token.c	×
create_new_postings_list()	token.c	×
db_get_token_id()	database.c	×
db_update_postings()	database.c	×
encode_postings()	postings.c	×
fetch_postings()	postings.c	×

merge_inverted_index()	postings.c	○
merge_postings()	postings.c	○
ngram_next()	token.c	○
text_to_postings_lists()	token.c	○
token_to_postings_list()	token.c	○
update_postings()	postings.c	○
wiser_is_ignored_char()	token.c	×

在 wiser 中，我们首先调用了函数 add_document()，该函数的作用是为文档的标题和正文构建倒排索引以及用于存储文档的数据库。

在函数 add_document() 内部会进行如下的处理。

① 从文档中取出词元

② 为每个词元建立倒排列表，并更新小倒排索引

③ 每当小倒排索引增长到一定大小，就将其与存储器上的倒排索引合并到一起

下面，我们就梳理一下函数 add_document() 的内部。

```
/**
 * 将文档添加到数据库中，建立倒排索引
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] title 文档的标题，为NULL时将会清空缓冲区
 * @param[in] body 文档正文
 */
static void
add_document(wiser_env *env, const char *title, const char *body)
```

```
{
  if (title && body) {
    UTF32Char *body32;
    int body32_len, document_id;
    unsigned int title_size, body_size;

    title_size = strlen(title);
    body_size = strlen(body);

    /* 将文档存储到数据库中并获取该文档对应的文档编号 */
    db_add_document(env, title, title_size, body, body_size); ❶
    document_id = db_get_document_id(env, title, title_size); ❷

    /* 转换文档正文的字符编码 */
    if (!utf8toutf32(body, body_size, &body32, &body32_len)) {
      /* 为文档创建倒排列表 */
      text_to_postings_lists(env, document_id, body32, body32_len,
                             env->token_len, &env->ii_buffer); ❸
      env->ii_buffer_count++;
      free(body32);
    }
    env->indexed_count++;
    print_error("count:%d title: %s", env->indexed_count, title);
  }

  /* 当缓冲区中存储的文档的数量达到了指定阈值时，更新存储器上的倒排索引 */
  if (env->ii_buffer &&
      (env->ii_buffer_count > env->ii_buffer_update_threshold || !title)) {
    inverted_index_hash *p;

    print_time_diff();

    /* 更新所有词元对应的倒排项 */
    for (p = env->ii_buffer; p != NULL; p = p->hh.next) {
      update_postings(env, p); ❺
    }
    free_inverted_index(env->ii_buffer);
    print_error("index flushed.");
    env->ii_buffer = NULL;
    env->ii_buffer_count = 0;

    print_time_diff();
  }
}
```

首先，我们将标题和正文存储到了用于存储文档的数据库中（❶）。

由于 SQLite 会自动为存储到数据库中的记录分配 ID，所以我们就把这个 ID 用作文档编号（❷）。

接下来，在（❸）的步骤中，我们通过调用函数 text_to_postings_lists()，并根据文档编号（document_id）和文档内容（body32），更新了存储在变量 env->ii_buffer 中的小倒排索引。

然后我们要判断是否需要合并索引（❹）。当 title 为 NULL 时，或者当已构建出小倒排索引的文档数量达到了阈值（env->ii_buffer_update_threshold）时，就合并索引。另外，title 为 NULL 还标志着所有的文档都已经处理完了。

env->ii_buffer_update_threshold 是一个阈值，决定了将多少个文档存储到小倒排索引中之后，就需要将小倒排索引与存储器上的倒排索引合并了。该阈值设定得越小，内存的使用量也就越小，但是这样会增加对存储器的访问次数。反过来，该阈值设定得越大，内存的使用量也就越大，但是这样能减少对存储器的访问次数。

最后在❺的步骤中我们通过调用函数 update_postings() 合并了倒排索引，并将合并后的结果写入数据库（存储器）中。

进一步阅读源代码

下面，让我们再详细地梳理一下在构建倒排索引的过程中先后被调用的几个函数吧。

▍函数 text_to_postings_lists()

该函数的作用是为文档编号和构成文档内容的字符串建立倒排列表的集合（倒排文件）。

```
/**
 * 为构成文档内容的字符串建立倒排列表的集合
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] document_id 文档编号。为0时表示把要查询的关键词作为处理对象
 * @param[in] text 输入的字符串
 * @param[in] text_len 输入的字符串的长度
 * @param[in] n N-gram中N的取值
 * @param[in,out] postings 倒排列表的数组（也可视作是指向小倒排索引的指针）。若
 * @retval 0 成功
```

```
 * @retval -1 失败
 */
int
text_to_postings_lists(wiser_env *env,
                       const int document_id, const UTF32Char *text,
                       const unsigned int text_len,
                       const int n, inverted_index_hash **postings)
{
  /* FIXME: now same document update is broken. */
  int t_len, position = 0;
  const UTF32Char *t = text, *text_end = text + text_len;

  inverted_index_hash *buffer_postings = NULL;

  for (; (t_len = ngram_next(t, text_end, n, &t)); t++, position++) { ❻
    /* 检索时，忽略掉由t中长度不足N-gram的最后几个字符构成的词元 */
    if (t_len >= n || document_id) {
      int retval, t_8_size;
      char t_8[n * MAX_UTF8_SIZE];

      utf32toutf8(t, t_len, t_8, &t_8_size); ❼

      retval = token_to_postings_list(env, document_id, t_8, t_8_size,
                                      position, &buffer_postings); ❽
      if (retval) { return retval; }
    }
  }

  if (*postings) { ❾
    merge_inverted_index(*postings, buffer_postings); ❿
  } else {
    *postings = buffer_postings;
  }

  return 0;
}
```

首先，我们通过调用位于 token.c 中的函数 ngram_next()，从字符串 t 中取出了一个 N-gram，同时还获取了词元的长度 t_len 和指向其首地址的指针 t（❻）。有关函数 ngram_next() 的讲解我们先放到后面，请继续阅读函数 text_to_postings_lists() 的代码。

接下来要做的是为由函数 ngram_next() 返回的每个词元创建倒排列表。在创建过程中，我们首先将词元的字符编码由 UTF-32 转换成了 UTF-

8（❼），随后通过调用函数 token_to_postings_list()，将该词元添加到倒排列表中（❽）。有关函数 token_to_postings_list() 的讲解我们也先放到后面。

让我们继续往下读，当❻的循环一结束，仅由传入本函数的文档构成的倒排索引也就构建出来了。

为了便于理解，我们再来看一个具体的例子。假设有这样一个文档，文档编号是 10，正文的内容是“自制搜索引擎”。将该文档传递给函数 text_to_postings_lists() 后，就可以构建出如图 3-2 所示的倒排索引。

构建出倒排索引以后，如果已经存在小倒排索引了（❾），就调用函数 merge_inverted_index() 将其与刚刚构建出的倒排索引合并（❿）。反之，如果还没有小倒排索引，就将刚刚构建出的倒排索引作为小倒排索引。有关函数 token_to_postings_list() 的讲解同样也先放到后面。



图 **3-2**　将文档“自制搜索引擎”传递给函数 **text_to_postings_lists()** 后构建出的倒排索引

前面我们从源代码级别上简单地梳理了一遍构建索引的流程，下面让我们再来梳理一下刚刚并未讲解的 4 个函数：ngram_next()、

token_to_postings_list()、update_postings() 和 merge_inverted_index()。

▍函数 **ngram_next()**

首先，我们来看一下在函数 text_to_postings_lists() 中被调用的函数 ngram_next()。该函数负责从字符串中取出 N-gram，返回词元的长度和词元首地址的指针。

```
/**
 * 将传入的字符串分割为N-gram
 * @param[in] ustr 输入的字符串（UTF-8）
 * @param[in] ustr_end 输入的字符串中最后一个字符的位置
 * @param[in] n N-gram中N的取值。建议将其设为大于1的值
 * @param[out] start 词元的起始位置
 * @return 分割出来的词元的长度
 */
static int
ngram_next(const UTF32Char *ustr, const UTF32Char *ustr_end,
           unsigned int n, const UTF32Char **start)
{
  int i;
  const UTF32Char *p;

  /* 读取时跳过文本开头的空格等字符 */
  for (; ustr < ustr_end && wiser_is_ignored_char(*ustr); ustr++) { ⓫
  }

  /* 不断取出最多包含n个字符的词元，直到遇到不属于索引对象的字符或到达了字符串的尾
  for (i = 0, p = ustr; i < n && p < ustr_end
       && !wiser_is_ignored_char(*p); i++, p++) { ⓬
  }

  *start = ustr;
  return p - ustr;
}
```

在读取构成词元的字符时，我们首先跳过了文本开头的空格等不属于索引对象的字符（⓫）。这里使用的是空循环，因此可以使用“;”作为 for 语句的循环体，但是为了易于理解，我们还是写成了空语句块“{}”的形式。函数 wiser_is_ignored_char() 的作用是判断给定的字符是否不属于检索对象。

接下来，通过⓬中的 for 语句，我们从文本中取出了 n 个字符。虽然这也是个空循环，但是循环条件并不简单，在循环时既要考虑不属于索引对象的字符，还要防止指针 p 超出字符串的末尾。

▍函数 token_to_postings_list()

下面再来看一下在函数 text_to_postings_lists() 中被调用的函数 token_to_postings_list()。函数 token_to_postings_list() 的作用是为文档中的一个词元创建倒排列表。

```
/**
 * 为传入的词元创建倒排列表
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] document_id 文档编号
 * @param[in] token 词元（UTF-8）
 * @param[in] token_size 词元的长度（以字节为单位）
 * @param[in] position 词元出现的位置
 * @param[in,out] postings 倒排列表的数组
 * @retval 0 成功
 * @retval -1 失败
 */
int
token_to_postings_list(wiser_env *env,
                       const int document_id, const char *token,
                       const unsigned int token_size,
                       const int position,
                       inverted_index_hash **postings)
{
  postings_list *pl;
  inverted_index_value *ii_entry;
  int token_id, token_docs_count;

  token_id = db_get_token_id(
               env, token, token_size, document_id, &token_docs_count); ⓭

  if (*postings) { ⓮
    HASH_FIND_INT(*postings, &token_id, ii_entry); ⓯
  } else {
    ii_entry = NULL; ⓰
  }

  if (ii_entry) { ⓱
    pl = ii_entry->postings_list; ⓲
    pl->positions_count++; ⓳
  } else {
```

```
    ii_entry = create_new_inverted_index(token_id,
                                document_id ? 1 : token_docs_count); ⑳
    if (!ii_entry) { return -1; }
    HASH_ADD_INT(*postings, token_id, ii_entry); ㉑

    pl = create_new_postings_list(document_id); ㉒
    if (!pl) { return -1; }
    LL_APPEND(ii_entry->postings_list, pl); ㉓
  }
  /* 保存位置信息 */
  utarray_push_back(pl->positions, &position); ㉔
  ii_entry->positions_count++; ㉕
  return 0;
}
```

首先，我们通过调用函数 db_get_token_id()，获取了词元对应的编号（⓭）。如果之前已将编号分配给了该词元，那么在此处获取的正是这个编号；反之，如果之前没有分配编号，那么函数 db_get_token_id() 会为该词元分配一个新的编号。

再往下看，如果存在已经构建好的小倒排索引（⓮），那么我们就从中获取关联到该词元编号上的倒排列表（⓯）。在获取时，我们以 token_id 为键调用了名为 HASH_FIND_INT() 的宏，并将从小倒排索引（postings）中获取到的倒排列表存储到变量 ii_entry（在内部实际上是 ii_entry->postings_list）中。

而如果找不到以 token_id 为键的倒排列表，那么就先将变量 ii_entry 的值设为 NULL（⓰）。

如果变量 ii_entry 的值不为 NULL，也就是说小倒排索引中存在关联到该词元上的倒排列表（⓱），那么我们就先将指针 pl 指向该倒排列表（⓲），然后再将该倒排列表中词元的出现次数增加 1（⓳）。在计算用于对检索结果进行排名的分数时，会用到词元的出现次数。

反之，如果变量 ii_entry 的值为 NULL，也就是说小倒排索引中不存在关联到该词元上的倒排列表，那么我们就先调用函数 create_new_inverted_index()，生成一个空的小倒排索引（⓴），然后再调用 HASH_ADD_INT()，将该词元添加到新建的小倒排索引中（㉑）。接下来，通过调用函数 create_new_postings_list()，我们创建出了仅由 1 个文档构成的倒排列表 pl（㉒），随后又将该倒排列表添加到

了刚刚生成的小倒排索引中（㉓）。

此时，指针 pl 指向关联到词元上的倒排列表。

接着，我们又通过调用函数 utarray_push_back()，将词元的出现位置添加到了倒排列表中存储着出现位置的数组的末尾（㉔）。

最后，我们又将当前词元在所有文档中的出现次数之和增加 1。出现次数之和的数据存储在关联到词元的倒排列表中（㉕）。

❚ 函数 **update_postings()**

下面，让我们再来看一下在函数 add_document() 中被调用的函数 update_postings() 吧。

```
/**
  * 将内存上（小倒排索引）的倒排列表与存储器上的倒排列表合并后存储
  * @param[in] env 存储着应用程序运行环境的结构体
  * @param[in] p 含有倒排列表的倒排索引中的索引项
  */
void
update_postings(const wiser_env *env, inverted_index_value *p)
{
  int old_postings_len;
  postings_list *old_postings;

  if (!fetch_postings(env, p->token_id, &old_postings,
                      &old_postings_len)) { ㉖
  buffer *buf;
  if (old_postings_len) {
    p->postings_list = merge_postings(old_postings, p->postings_list); ㉗
    p->docs_count += old_postings_len;
  }
  if ((buf = alloc_buffer())) { ㉘
    encode_postings(env, p->postings_list, p->docs_count, buf); ㉙
    db_update_postings(env, p->token_id, p->docs_count,
                       BUFFER_PTR(buf), BUFFER_SIZE(buf)); ㉚
    free_buffer(buf);
  }
  } else {
    print_error("cannot fetch old postings list of token(%d) for update.",
                p->token_id);
  }
}
```

首先，我们通过调用函数 fetch_postings()，从存储器中取出了作为合并源的倒排列表（㉖）。

如果存储器中存在作为合并源的倒排列表，那么就调用函数 merge_postings()，将该倒排列表和要合并进来的倒排列表（p->postings_list）合并在一起（㉗）。有关函数 merge_postings() 的讲解先放到后面，我们继续往下看。

接下来，我们申请了一块临时的缓冲区（㉘），利用这块缓冲区和函数 encode_postings()，将内存上的倒排列表转换成了字节序列（㉙）。

最后，我们又通过调用函数 db_update_postings()，将转换后的字节序列存储到了存储器中（㉚）。

❙ 函数 **merge_inverted_index()**

下面我们再来看一下在函数 text_to_postings_lists() 中被调用的函数 merge_inverted_index()。函数 merge_inverted_index() 的作用是合并内存上的两个倒排索引。

```
/**
  * 合并两个倒排索引
  * @param[in] base 合并后其中的元素会增多的倒排索引（合并目标）
  * @param[in] to_be_added 合并后就被释放的倒排索引（合并源）
  *
  */
void
merge_inverted_index(inverted_index_hash *base,
                     inverted_index_hash *to_be_added) ㉛
{
  inverted_index_value *p, *temp;

  HASH_ITER(hh, to_be_added, p, temp) { ㉜
    inverted_index_value *t;
    HASH_DEL(to_be_added, p); ㉝
    HASH_FIND_INT(base, &p->token_id, t); ㉞
    if (t) { ㉟
      t->postings_list = merge_postings(t->postings_list, p->postings_list)
      t->docs_count += p->docs_count; ㊲
      free(p);
```

```
    } else {
      HASH_ADD_INT(base, token_id, p); ㊳
    }
  }
}
```

该函数会先接收两个内存上的倒排索引作为参数（㉛），然后将合并源（传递给第 2 个参数的倒排索引）中的内容合并到目标（传递给第 1 个参数的倒排索引）中。合并完成后，该函数还会从内存上释放出传递给第 2 个参数的倒排索引所占用的空间。

具体的合并方法是，先将存储在作为合并源的倒排索引中的所有倒排列表逐一取出，存到临时变量里（㉜），然后再将刚刚取出的倒排列表从作为合并源的关联数组中删除（㉝）。

接下来，用刚取出的倒排列表所对应的词元，到合并目标中去查找与该词元对应的倒排列表（㉞）。如果合并目标中存在相应的倒排列表（㉟），就调用函数 merge_postings()，将合并源和合并目标中的元素所带有的倒排列表合并在一起（㊱），并将出现过该词元的文档数相加（㊲）——有关函数 merge_postings() 的实现我们稍后再讲解——反之，如果合并目标中没有相应的倒排列表，就将获取的合并源中的倒排列表直接添加到作为合并目标的关联数组中（㊳）。

另外，在这里，我们通过使用 uthash 提供的宏 HASH_ITER()，实现了将合并源中的元素逐一取出的循环。

▍函数 merge_postings()

最后，我们再来详细地看一下在函数 merge_inverted_index() 和函数 update_postings() 中都调用了的函数 merge_postings() 吧。

```
static postings_list *
merge_postings(postings_list *pa, postings_list *pb) ㊴
{
  postings_list *ret = NULL, *p; ㊵
  /* 用pa和pb分别遍历base和to_be_added（参见函数merge_inverted_index）中的倒排
  while (pa || pb) { ㊶
    postings_list *e;
    if (!pb || (pa && pa->document_id <= pb->document_id)) { ㊷
      e = pa; ㊸
```

```
      pa = pa->next; ㊹
    } else if (!pa || pa->document_id >= pb->document_id) { ㊺
      e = pb; ㊻
      pb = pb->next; ㊼
    } else {
      abort();
    }
    e->next = NULL; ㊽（以下7行）
    if (!ret) {
      ret = e;
    } else {
      p->next = e;
    }
    p = e;
  }
  return ret;
}
```

该函数会接收两个内存上的倒排列表作为参数（㊴），并返回将其合并后的倒排列表。我们使用变量 ret 来管理合并后的倒排列表（㊵）。

在分别使用变量 pa 和 pb 遍历两个倒排列表中的元素（倒排项）的过程中，我们要不断地从 pa 和 pb 所指向的两个元素中挑选出文档编号较小的一方，然后将其添加到合并后的倒排列表中。

如果 pa 所指向的文档编号小于 pb 所指向的文档编号（㊷），那么就将 pa 所指向的元素添加到合并后的倒排列表中（㊸），并让 pa 指向下一个元素（㊹）。

反之，如果 pb 所指向的文档编号小于 pa 所指向的文档编号（㊺），那么就将 pb 所指向的元素添加到合并后的倒排列表中（㊻），并让 pb 指向下一个元素（㊼）。

另外，在实际的合并过程中，在（㊸）和（㊻）两处，我们并没有将要添加的元素直接添加到倒排列表中，而是先将其保存到了变量 *e* 中。到了后面㊽的那一段代码，我们会将变量 *e* 添加到合并后的倒排列表中。如果 pa 和 pb 双方都指向了各自倒排列表中最后一个元素之后的位置，那么合并处理就此结束（㊶）。

至此为止，我们就梳理完了以函数 add_document() 为入口的构建倒排索引的处理流程。下面我们再来简单地回顾一下整个处理流程吧。

① 从文档中取出词元。

② 为每个词元创建倒排列表并将该倒排列表添加到小倒排索引中。

③ 每当小倒排索引增长到一定大小，就将其与存储器上的倒排索引合并到一起。

专栏

根据实际情况设计搜索引擎（系统）

虽然借助倒排索引可以加快全文搜索的速度，但是构建倒排索引却要花费大量的时间。例如，当遇到“虽然检索次数很少，但是需要经常检索最新的信息”这种情况时，有时不建立索引，而是每次都使用 grep 等工具对文档本身进行全扫描，反而能够得到更好的效果。再比如，当要检索的单词有一定范围时，只需将构建倒排索引的对象限定在特定的单词上，就既能降低构建倒排索引时的开销，又能加快那些使用了倒排索引的检索速度。

综上所述，我们应该根据实际情况灵活地调整搜索引擎（系统）的构成和设计方案。例如，Twitter 在实现倒排索引时就做出过这样一个改进：由于在 Twitter 上用户最多只能发表 140 个字的推文，所以只需要 1 个字节就可以存储词元的出现位置信息了，Twitter 就利用这个特征，节约了倒排列表所需的存储空间。这就是一个根据实际情况提升索引结构使用效率的例子。

第 4 章　开始检索吧

在第 3 章中，我们看到了倒排索引的实现过程以及对其进行构建处理的过程。进入本章，就让我们再来了解一下如何对已构建的倒排索引进行检索吧。检索处理的基础知识也已经在第 1 章讲解过了，因此在阅读时若有疑问，不妨再回过头去读一读第 1 章。另外，由于在第 3 章我们并没有对已构建的倒排索引进行压缩，所以在本章也还是以未进行压缩的倒排索引为前提继续讲解。我们将在第 5 章统一讲解有关压缩倒排索引的内容。

4-1 检索处理的大致流程

在本节，我们先来简单地复习一下在第 1 章讲解过的有关检索处理的流程。

充分理解检索处理的流程

让我们举例来说。假设搜索引擎接收到了内容为“自制搜索引擎”的查询，那么接下来就会进行如下的检索处理。由于 wiser 只支持 AND 检索，所以在下面的讲解中也假设进行的是 AND 检索。

① 将查询分割为词元（如果使用的是 bi-gram，那么就会分割出“自制”“制搜”“搜索”“索引”“引擎”5 个词元）。

② 将分割出的各个词元，按照出现过该词元的文档数量进行升序排列（升序排列的理由将在稍后阐明）。

③ 获取各个词元的倒排列表，并从中取出文档编号和该词元在文档中出现位置的列表。

④ 如果所有词元都出现在同一个文档中，并且这些词元的出现位置都是相邻的，那么就将该文档添加到检索结果中。

⑤ 计算已添加到检索结果中的各文档与查询的匹配度（在 wiser 中，我们使用 TF-IDF 值作为匹配度）。

⑥ 将检索结果按照匹配度的降序排列。

⑦ 从经过排序的检索结果中取出排在前面的若干个文档作为检索结果返回。

另外，在第②步中，之所以要将分割出的各个词元按照出现过该词元的文档数量进行升序排列，是因为这样做可以尽早缩小检索结果的范围。而尽早缩小检索结果的范围，就可以减少在步骤④中进行的比较处理的次数。

另外，与构建倒排索引时一样，在上述流程中，我们通过一次错开 1 个

字符的方式将查询分割为了 bi-gram 的词元。但是，该方法未必是个高效的方法。因为原本无需每次错开 1 个字符才能将查询分割为 bi-gram 的词元，而是只需要将查询分割为无重复部分的词元即可。例如，在检索“自制搜索引擎”时，我们可以先将查询分割为“自制”“搜索”“引擎”3 个词元，然后分别获取它们各自的倒排列表，最后使用这些倒排列表进行文档匹配的判定。不过在本章中，我们先不讲解这种处理方法如何实现，而是在第 6 章“挑战 wiser 的优化及参数的调整”中重新提及该话题，请诸位到时务必挑战一下这种处理方法。

4-2　使用倒排索引进行检索

下面就让我们从源代码级别梳理一下使用倒排索引进行检索处理的流程吧。

从源代码级别梳理检索处理的流程

以检索模式启动 wiser 后，函数 search() 就会被调用。下面我们就从该函数的内部开始梳理。

```
/**
 * 进行全文搜索
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] query 查询
 */
void
search(wiser_env *env, const char *query)
{
  int query32_len;
  UTF32Char *query32;

  if (!utf8toutf32(query, strlen(query), &query32, &query32_len)) { ❶
    search_results *results = NULL;

    if (query32_len < env->token_len) { ❷
      print_error("too short query.");
    } else {
      query_token_hash *query_tokens = NULL;
      split_query_to_tokens(
        env, query32, query32_len, env->token_len, &query_tokens); ❸
      search_docs(env, &results, query_tokens); ❹
    }

    print_search_results(env, results); ❺

    free(query32);
  }
}
```

首先，我们将查询字符串的编码由 UTF-8 转换成了 UTF-32（❶）。

随后判断了查询字符串的长度是否大于 N-gram 中 N 的取值（❷）。如果长度大于 N，就调用函数 split_query_to_tokens()，将词元从查询字符串中提取出来（❸）。我们先继续往下看，稍后再讲解该函数。

接下来，以刚刚提取出来的词元作为参数，调用函数 search_docs()，开始进行检索处理（❹）。有关函数 search_docs() 的细节，我们将在稍后梳理。

最后在❺的步骤中，调用了用于打印检索结果的函数 print_search_results()。至此，检索处理的流程就结束了。函数 print_search_results() 会先将检索结果从 results 中逐一取出，然后以检索结果中的文档编号为查询条件，从文档数据库中取出相应的文档标题，最后输出获取到的标题和检索的得分。

解读 **split_query_to_tokens()** 函数的具体实现

在了解了大致的检索流程后，我们先来看一下用于将查询字符串转换为词元序列的函数 split_query_to_tokens()。

```
/**
 * 从查询字符串中提取出词元的信息
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] text 查询字符串
 * @param[in] text_len 查询字符串的长度
 * @param[in] n N-gram中N的取值
 * @param[in,out] query_tokens 按各词元编号存储位置信息序列的关联数组
 *                             若传入的是指向NULL的指针，则新建一个关联数组
 * @retval 0 成功
 * @retval -1 失败
 */
int
split_query_to_tokens(wiser_env *env,
                      const UTF32Char *text,
                      const unsigned int text_len,
                      const int n, query_token_hash **query_tokens)
{
  return text_to_postings_lists(env,
                                0, /* 将document_id设为0 */
                                text, text_len, n,
                                (inverted_index_hash **)query_tokens);
}
```

从源代码可以看出，该函数实际上就是又去调用了在第 3 章讲解过的函数 text_to_postings_lists()。之所以这样做，是因为从字符串中生成倒排列表的处理过程，和从查询中提取出词元后再取出各词元位置信息的处理过程有很多共通之处。同时，这样做也是为了使讲解的内容更易于理解。

另外，在检索时，我们将 0 传递给了函数 text_to_postings_lists() 的第 2 个参数，以表示不需要使用文档编号。也就是说，在函数 text_to_postings_lists() 的内部，是根据文档编号是否为 0 来判别当前是构建模式还是检索模式的。

同样地，由于用于管理查询中词元的结构与用于构建索引的结构体非常相似，所以我们通过为后者赋予别名的方式，定义出了表示前者的结构体。

```
typedef inverted_index_hash query_token_hash;
typedef inverted_index_value query_token_value;
typedef postings_list token_positions_list;
```

使用具体示例加深对检索处理流程的理解

下面，我们再来稍微详细地看一下作为检索处理核心的函数 search_docs()，以及在该函数中被调用的函数 search_phrase()。由于这两个函数的实现都有些复杂，所以会先通过具体的示例粗略地讲解一下，然后再来梳理其源代码。

在本例中，假设我们要使用查询字符串“搜索引擎”对如图 4-1 所示的倒排索引进行检索。



图 **4-1**　在具体示例中使用的（部分）倒排索引

首 先，我们将查询字符串分割成了 3 个词元 ——“搜索 ”“索引”“引擎”。然后，分别扫描关联到这 3 个词元上的倒排列表。如果能够找到一个在所有倒排列表中都出现过的文档编号，那么就将它所指向的文档

加入到候选检索结果中。

在这个过程中，我们要将第一个词元“搜索”称为“词元 A”。到后面真正开始阅读函数 search_docs() 的代码时，我们还会用到这个叫法。

具体的检索流程如下。

首先，从词元 A 的倒排列表的开头获取文档编号为 15 的元素；然后，在另外两个词元“索引”和“引擎”的倒排列表中，检查是否也存在文档编号为 15 的元素。

在词元“索引”的倒排列表中，第一个文档编号是 13，第二个文档编号是大于 15 的 17。此时就可以确定，文档编号小于 17 的文档一定不属于候选检索结果。之所以这样说，是因为倒排列表是按照文档编号的升序排列的。既然词元 A“搜索”未曾出现在编号小于 15 的文档中，词元“索引”也未曾出现在编号大于 15 且小于 17 的文档中，那么，在编号小于 17 的文档中就一定不可能同时出现“搜索”和“索引”。

由于编号小于 17 的文档不可能成为检索结果，所以会继续向后读取词元 A 的倒排列表，直到发现某个文档编号不小于 17 的元素为止。继续向后读，就读取到了文档编号为 18 的元素。

至此为止，我们做了如下几件事。

- 首先获取了词元 **A** 的文档编号，然后检查了其他的词元是否也带有相同的文档编号

- 如果没有发现带有相同文档编号的词元，那么接下来就继续向后读取词元 **A** 的倒排列表，直到遇到更大的文档编号为止

下一次循环时，词元 A 的文档编号是 18。对于除词元 A 以外的其他词元，只需要继续向后读取其各自的倒排列表，就可以知道是否包含着文档编号为 18 的元素了，或者说就可以知道哪个倒排列表中含有编号为 18 的文档了。由于在本例中所有的倒排列表中都含有编号为 18 的文档，所以该文档就成为了候选检索结果。

但为什么编号为 18 的文档不是正式检索结果，而只是候选检索结果呢？这就需要诸位回忆一下在第 1 章中讲解过的有关短语检索的知识了。例

如，假设我们将内容为“不可能”的查询发送给了搜索引擎，虽然有些文档也同时包含了“不可”和“可能”，但是这些文档却未必包含连在一起的“不可能”3 个字。以“他不可一世的态度可能源于他童年时的经历”这句话为例，虽然先后包含了“不可”和“可能”这两个词，但是却并没有包含“不可能”这个短语。因此要想查找像是“不可能”这样的短语，就必须确认“不可”和“可能”是不是相邻出现的。

也就是说，在倒排列表中，我们只需要关注 3 个含有文档编号 18 的方框中的出现位置，并检查“搜索”“索引”“引擎”这 3 个词元是不是顺序（相邻）出现的即可。

换句话说，就是以“搜索”的出现位置为起始位置，确认“索引”的出现位置是否为起始位置 + 1，“引擎”的出现位置是否为起始位置 + 2。在本例中，“搜索”的出现位置是 30、“索引”是 31、“引擎”是 32，因此满足相邻出现的条件。记录在编号为 18 的文档中发现了短语后，就可以继续向后处理词元 A 的倒排列表了。

接下来要做的只不过是反复执行上述处理。

继续向后扫描，使词元 A 的文档编号依次变为 30 和 213。当文档编号为 213 时，由于在词元“索引”的倒排列表中，没有比 213 更大的文档编号了，同时也无法再继续向后读取下一个元素了，所以检索处理至此结束。

最后，对找到的检索结果进行排序（由于在本例中只有 1 条检索结果，所以也就没有排序的必要了）。

至此为止，我们通过具体的示例，简单地介绍了稍后将要讲解的函数 search_phrase() 和函数 search_docs() 的处理流程。如果感到头脑有些混乱，不妨试着先在纸上梳理一下上述处理过程。

解读函数 **search_docs()** 的实现细节

下面我们来看一下函数 search_docs() 的具体实现吧。

```
/**
 * 检索文档
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in,out] results 检索结果
```

```
 * @param[in] tokens 从查询中提取出来的词元信息
 */
void
search_docs(wiser_env *env, search_results **results,
            query_token_hash *tokens)
{
  int n_tokens;
  doc_search_cursor *cursors;

  if (!tokens) { return; }

  /* 按照文档频率的升序对tokens排序 */
  HASH_SORT(tokens, query_token_value_docs_count_desc_sort); ❻

  /* 初始化 */
  n_tokens = HASH_COUNT(tokens);
  if (n_tokens &&
      (cursors = (doc_search_cursor *)calloc(
                  sizeof(doc_search_cursor), n_tokens))) { ❼
    int i;
    doc_search_cursor *cur;
    query_token_value *token;
    for (i = 0, token = tokens; token; i++, token = token->hh.next) { ❽
      if (!token->token_id) { ❾

        /* 当前的token在构建索引的过程中从未出现过 */
        goto exit;
      }
      if (fetch_postings(env, token->token_id,
                         &cursors[i].documents, NULL)) { ❿


        print_error("decode postings error!: %d\n", token->token_id);
        goto exit;
      }
      if (!cursors[i].documents) {
        /* 虽然当前的token存在，但是由于更新或删除导致其倒排列表为空。*/
        goto exit;
      }
      cursors[i].current = cursors[i].documents; ⓫

    }
    while (cursors[0].current) { ⓬

      int doc_id, next_doc_id = 0;
      /* 将拥有文档最少的词元称作A。 */
      doc_id = cursors[0].current->document_id; ⓭
```

```
    /* 对于词元A以外的词元，不断获取其下一个document_id，直到当前的document_i
    for (cur = cursors + 1, i = 1; i < n_tokens; cur++, i++) { ⓮

      while (cur->current && cur->current->document_id < doc_id) { ⓯

        cur->current = cur->current->next; ⓰
      }
      if (!cur->current) { goto exit; } ⓱

      /* 对于词元A以外的词元，如果其document_id不等于词元A的document_id，那么
      if (cur->current->document_id != doc_id) { ⓲

        next_doc_id = cur->current->document_id; ⓳

        break;
      }
    }
    if (next_doc_id > 0) { ⓴

      /* 不断获取A的下一个document_id，直到其当前的document_id不小于next_doc
      while (cursors[0].current
             && cursors[0].current->document_id < next_doc_id) { ㉑
        cursors[0].current = cursors[0].current->next; ㉒
      }

    } else {

      int phrase_count = -1;
      if (env->enable_phrase_search) {
        phrase_count = search_phrase(tokens, cursors); ㉓
      }
      if (phrase_count) {
        double score = calc_tf_idf(tokens, cursors, n_tokens,
                                   env->indexed_count); ㉔

        add_search_result(results, doc_id, score); ㉕
      }
      cursors[0].current = cursors[0].current->next; ㉖

    }
  }
exit: ㉗

  for (i = 0; i < n_tokens; i++) {
    if (cursors[i].documents) {
      free_token_positions_list(cursors[i].documents);
```

```
      }
    }
    free(cursors);
  }
  free_inverted_index(tokens);

  HASH_SORT(*results, search_results_score_desc_sort); ㉘
}
```

首先，要对从查询字符串中取出的词元（token）集合，按照各词元文档频率的升序进行排列（❻）。文档频率是指在作为检索对象的所有文档中，出现过某个词元的文档数量。这样排序的理由已经在本章的开头解释过了。

接下来，我们为每个词元都分配了一块内存空间，用于存储表示当前指向了哪个文档的状态（游标）（❼）。稍后遍历每个词元对应的倒排列表时，会用到这些游标。

在❽的步骤中，我们从词元集合中将词元逐一取出。此时，如果某个词元还没有被分配编号（❾），则说明无法从数据库中获取到该词元的编号，也就是说查询该词元得到的结果为空。一旦出现了这种情况，我们就跳转到 exit 标签以中断检索处理。而如果词元已经被分配了编号（❿），那么接下来我们就通过调用函数 fetch_postings()，从索引（数据库）中获取该词元对应的倒排列表。此时，如果能够正确地获取到倒排列表，我们就设置该词元对应的游标，使其指向倒排列表中的第一个文档（⓫）。

至此，我们通过直到（⓫）为止的一系列处理，设定好了各个词元所对应的游标。接下来，和之前讲解具体示例时一样，我们将第一个词元称作词元 A。

在接下来的检索处理（标有⓬的循环）中，我们通过不断向后移动各个游标，来查找在所有的倒排列表中共同出现的文档编号。只要还没有扫描到词元 A 所对应的倒排列表的末尾，检索处理就不会结束。的确，我们应该一一检查所有词元的扫描位置，看看有没有扫描到各自倒排列表的末尾，但是在这里，只需要检查对词元 A 的扫描是否结束了即可。

在该循环中，我们首先通过词元 A 的游标获取到了一个文档编号

（⓭）。我们称这个文档编号为“文档编号 A”。

接下来，需要检查除词元 A 以外的所有词元的倒排列表，看看其中 是否也包含文档编号 A（⓮）。检查时，如果游标指向的文档编号小于文档编号 A（⓯），那么就使游标指向下一个文档编号（⓰）。

在扫描除词元 A 以外的词元的倒排列表的过程中，如果游标到达了该倒排列表的末尾，就要强制中断检索处理（⓱）。

在⓲的步骤中，如果当前游标所指向的文档编号不等于文档编号 A，那么说明在当前游标所指向的词元的倒排列表中不存在文档编号 A，同时也说明当前游标所指向的文档编号大于文档编号 A。因此，我们就将当前游标所指向的文档编号赋值给变量 next_doc_id（⓳）。

如果 next_doc_id 大于 0（⓴），即文档编号 A 所指向的文档并不属于候选检索结果时，要将词元 A 的游标不断向后移动，直到该游标所指向的文档编号不小于 next_doc_id 为止（㉑、㉒）。通过反复进行上述操作，最终就能达到所有的游标都指向同一个文档编号的状态。

当 next_doc_id 等于 0 时，说明文档编号 A 所指向的文档成为了候选检索结果，因此随后还要通过函数 search_phrase() 来确认作为短语的查询字符串是否存在（㉓）。如果确实存在短语，那么就调用函数 calc_tf_idf()，计算出用于排序的得分（㉔），并调用函数 add_search_result()，将文档编号添加到检索结果的列表中（㉕）。

接下来继续向后移动词元 A 的游标（㉖），并开始重复执行之前的处理过程。

检索一结束，exit 标签后面的语句就会被执行（㉗）。在执行的过程中，既会回收分配给所有词元的、用来存储检索信息的存储空间，又会回收由查询字符串生成的词元集合的信息。最后，在（㉘）的步骤中，我们对检索结果按照得分的高低进行了降顺排列。

解读函数 **search_phrase()** 的实现

下面，我们再来看一下用于短语检索的函数 search_phrase()。不过，在深入阅读源代码之前，需要先稍微了解一下处理短语检索的策略。

假设文档中存在短语，此时构成短语的词元出现的位置如下所示。

11、12、13、14

86、87、88、89

从各出现位置中减去其在短语内的相对位置后，得到的结果如下所示。

- 当出现位置依次为 **11、12、13、14** 时

  → **11、11、11、11**

  ※ **11 - 0 = 11，12 - 1 = 11，13 - 2 = 11，14 - 3 = 11**

- 当出现位置分别为 **86、87、88、89** 时

  → **86、86、86、86**

  ※ **86 - 0 = 86，87 - 1 = 86，88 - 2 = 86，89 - 3 = 86**

由此可以看出，此时所有的出现位置都是相等的。通过这样的减法运算，就可以将检索短语的处理过程转化为“查找所有词元共同出现的位置”这一处理过程了。这与负责查找所有词元同时出现的文档编号的函数 search_docs() 所进行的处理非常相似，所以说，其实在函数 search_phrase() 中进行和函数 search_docs() 同样的处理就可以了。

下面，我们就来梳理一下源代码。

```
/**
 * 进行短语检索。
 * @param[in] query_tokens 从查询中提取出的词元信息
 * @param[in] doc_cursors 用于检索文档的游标的集合
 * @return 检索出来的短语数量
 */
static int
search_phrase(const query_token_hash *query_tokens,
              doc_search_cursor *doc_cursors)
{
  int n_positions = 0;
  const query_token_value *qt;
  phrase_search_cursor *cursors;
```

```
/* 获取查询中词元的总数 */
for (qt = query_tokens; qt; qt = qt->hh.next) { ㉙
  n_positions += qt->positions_count;
}

if ((cursors = (phrase_search_cursor *)malloc(sizeof(
                 phrase_search_cursor) * n_positions))) { ㉚
  int i, phrase_count = 0;

  phrase_search_cursor *cur;
  /* 初始化游标 */
  for (i = 0, cur = cursors, qt = query_tokens; qt;
       i++, qt = qt->hh.next) { ㉛
    int *pos = NULL;
    while ((pos = (int *)utarray_next(qt->postings_list->positions,
                                      pos))) { ㉜
      cur->base = *pos; ㉝
      cur->positions = doc_cursors[i].current->positions;
      cur->current = (int *)utarray_front(cur->positions); ㉞
      cur++;
    }
  }
  /* 检索短语 */
  while (cursors[0].current) { ㉟
    int rel_position, next_rel_position;
    rel_position = next_rel_position = *cursors[0].current -
                                       cursors[0].base; ㊱

    /* 对于除词元A以外的词元，不断地向后读取出现位置，直到其偏移量不小于词元A的
    for (cur = cursors + 1, i = 1; i < n_positions; cur++, i++) { ㊲
      for (; cur->current
           && (*cur->current - cur->base) < rel_position;
           cur->current = (int *)utarray_next(cur->positions, cur->curren
{} ㊳
      if (!cur->current) { goto exit; } ㊴

      /* 对于词元A以外的词元，若其偏移量不等于A的偏移量，就退出循环 */
      if ((*cur->current - cur->base) != rel_position) { ㊵
        next_rel_position = *cur->current - cur->base; ㊶
        break;
      }
    }
    if (next_rel_position > rel_position) { ㊷
      /* 不断向后读取，直到词元A的偏移量不小于next_rel_position为止 */
      while (cursors[0].current &&
             (*cursors[0].current - cursors[0].base) < next_rel_position)
        cursors[0].current = (int *)utarray_next(
```

```
                            cursors[0].positions, cursors[0].current); ㊹
        }
      } else {
        /* 找到了短语 */
        phrase_count++; ㊺
        cursors->current = (int *)utarray_next(
                            cursors->positions, cursors->current); ㊻
      }
    }
exit:
    free(cursors);
    return phrase_count; ㊼
  }
  return 0;
}
```

首先，我们统计出了查询中的词元总数（㉙），并为查询中的每个词元都分配了一个结构体，用来表示用于短语检索的游标（㉚）。在初始化这些游标的过程中（㉛，㉜），会将词元在查询中的出现位置存储到 cur->base 中（㉝），将对词元在文档中出现位置的引用存储到 cur->current 中（㉞）。

查找短语的处理过程是在标有㉟的循环中进行的。我们沿用在讲解函数 search_docs() 时用到的术语，依然称第一个词元为词元 A。不同的是，在该函数中查找的对象是出现位置的列表，而不是文档编号的列表。因此在该函数中使用游标管理的是“在关联着词元和文档编号的出现位置列表中，当前指向的是哪个位置”。第 i 个词元的游标存储在 cursors[i].current 中。

正如前文所述，对于所有的词元，都要从头检查其在文档内的出现位置。与通过函数 search_docs() 查找带有相同文档编号的词元类似，在该函数中要查找的是，从各词元的出现位置中减去其在查询中的出现位置后得到的偏移量完全相同的词元。

具体来说，就是在㊱中用词元 A 的出现位置减去其在查询中的出现位置，得到词元 A 的偏移量，然后在㊲的循环中，对除词元 A 以外的所有词元逐一进行相同的减法运算。每完成一次减法运算，都要检查所得到的偏移量是否和词元 A 的偏移量相等。如果不相等，则说明短语不存在，因此要重新对变量 next_rel_position 赋值，以便跳过没有必要处理的词元的出现位置（㊳、㊴、㊵、㊶）。

当找不到短语时（㊷），由于小于 next_rel_position 的偏移量都已经搜索过了，所以我们要向后移动词元 A 的游标，直到其偏移量超过 next_rel_position 为止（㊸、㊹）。

如果找到了短语，那么就将用于存储短语出现次数的变量 phrase_count 的值加 1（㊺），并使词元 A 的游标指向其下一个出现位置（㊻）。

由于此时已经知道了短语是否存在，所以也可以就此结束处理，但是考虑到短语的出现次数还会用在后面的评分处理等环节，因此我们还是决定让处理继续进行下去。

通过反复进行上述循环，即可找出全部的短语。最后，在返回了找到的短语个数后，就可以结束处理了（㊼）。

至此，我们就梳理完了使用倒排索引进行全文搜索处理的流程。如果感到难以理解，不妨先将下面的大致流程记在心中，然后再试着梳理源代码。也可以边梳理边在纸上将处理流程和变量的值写出来。

① 将查询分割成词元。

② 获取每个词元的倒排列表。

③ 从多个倒排列表中查找匹配查询的文档（即在带有相同文档编号的倒排项中，判断位置信息是否是相邻的），并将找到的文档添加到作为检索结果的文档集合中。

④ 计算作为检索结果的文档的得分，并基于该得分对结果排序。

专栏

如何实现标签检索

在各式各样的 Web 服务中，通常都会提供一种称作“标签”的功能。以Twitter 为例，在作为留言的推文中，“#”后面的字符串就被称为主题标签（Hashtag）。在 Twitter 中，用户可以看到带有同一主题标签的留言列表。像这样的标签检索，通常都是通过事先将带有特定标签的文档编号集合存储到表中，然后使用该表获取带有同一标签的文档编号集合来实现的。

敏锐的读者也许已经注意到了，这种实现方案其实就是倒排索引。由于可以将标签视为一种词元，所以只要借助倒排索引，就可以加快标签检索的速度了。另外，由于实现标签检索时不需要考虑出现位置的相关信息，所以使用文档级别的倒排文件就足够了。

第 **5** 章　压缩倒排索引

5-1　压缩的基础知识

压缩倒排索引的好处

在使用倒排索引进行检索的过程中，总检索时间中的大部分时间往往花费在了从二级存储读取倒排索引上。于是，就经常可以看到在存储倒排索引前，对其进行压缩以减少从二级存储读取的时间，进而使检索处理得以高速运转的对策。

也就是说，我们可以根据如下原理，通过压缩倒排索引来加快检索处理的速度。

从二级存储中读取（部分）经过压缩的倒排索引的时间

＋还原倒排索引的时间

＜

从二级存储中读取（部分）尚未经过压缩的倒排索引的时间

在本节，就让我们详细地看一下有关压缩倒排索引的方法吧。

专栏

压缩的目的

说到压缩技术，特别是在存储空间很紧张的年代，其主要目的就是减少存储空间的使用。但是，对于作为当今主流的二级存储装置磁盘驱动器而言，其单位容量的价格已经非常低廉了。可以说我们已经迎来了只需 500 元左右就可以购买到存储容量为几 TB 的磁盘的时代了。存储容量也正在逐渐转变为随手可得的资源。因此，在近几年，使用压缩技术的目的也正逐渐由“减少存储空间的使用”转变为“实现检索的高速化”。在检索时，为了填补处理器和磁盘驱动器

在速度上的差距，通常都会对索引进行压缩。也就是说，可以将压缩处理看作是一种分担负载尝试，通过减少从二级存储读出的数据量，以及额外进行的相应还原处理，即可将检索处理中集中在二级存储上的部分负载转移到处理器中。

倒排索引的压缩方法

倒排索引的压缩分为针对词典的压缩和针对倒排文件的压缩两种。

我们可以通过使用更少的信息量表示单词的集合来实现词典的压缩。例如，对于按照词典顺序排列的单词列表而言，通过避免重复存储相同的前缀，就可以减少存储词典时所需的必要存储空间。但是，在大多数情况下，由于词典的大小远远小于倒排文件的大小，所以一般认为压缩词典对于加快检索处理的速度并没有太大的贡献。因此，在本书中也不会过多地介绍压缩词典的方法。有关压缩词典的详细资料，可以参阅参考文献 3、4。

而倒排文件的压缩，可以通过使用更少的信息量表示其构成要素来实现。构成要素就是指文档编号、单词在文档内的出现次数（TF，Term Frequency，词频）以及由单词在文档内的偏移量构成的整数数组。下面，就让我们详细地看一下压缩倒排文件的方法。

倒排文件的压缩方法

在一般的程序中，大多数情况下都会为整数分配 4 或 8 个字节等定长的编码，但是在处理倒排文件时，由于经常要处理大量数值较小的整数，所以为了使用更少的信息量来表示整数，通常都会采用长度可变而非固定的编码方式。另外，由于倒排文件中的整数序列通常都是按照文档编号或文档内偏移量的升序排列的，所以对于这样的整数序列，一旦计算出了前后两个整数的差值，就可以使用更小的数值（更少的信息量）来表示其中的整数了。也就是说正如预期的那样，使用可变长度的编码确实可以带来大幅度的压缩。在本节，我们会介绍几种具有代表性的、用于对倒排文件进行编码的可变长编码。

▍**unary** 编码

unary 编码是一种简单直观的编码方法，对于整数 x（$x \geq 0$）来说， 只需要用 x 个“1”和 1 个“0”即可表示这个整数。以十进制数的 10 为例，

它的 unary 编码就是“11111111110”。而当 $x = 0$ 时，其 unary 编码为 0。

■ **gamma** 编码和 **delta** 编码

在 gamma 编码中，我们首先要将整数 x（$x > 0$）分解为 $2^e + d$（$e = \log_2 x$，$0 \leq d < 2^e$）的形式，然后用 unary 编码表示 $e + 1$，用比特宽度为 e 的二进制编码 [1] 表示 d。

> [1]所谓二进制编码，就是一种用二进制数字 0 和 1 表示数据的方式。这种方式也是计算机内部标准的数据存储方式。——译者注

还是以整数 10 为例，由于 $e = \log_2 10 \approx 3$，所以把 $d = 10 - 2^3 = 2$ 的二进制编码“010”接在 $e + 1 = 3 + 1 = 4$ 的 unary 编码“11110”之后，就可以得到 10 的 gamma 编码，即“11110010”。

而 delta 编码是一种把在 gamma 编码中用 unary 编码的部分再次进行 gamma 编码的编码方式。

■ **variable-byte** 编码（**byte-aligned** 编码）

variable-byte 编码是一种用由多个字节构成的序列表示整数的编码方式。在对整数编码时，各个字节的构成如下所示。

- 用最右侧的 **7** 个比特表示数值
- 用最左侧的 **1** 个比特表示是否需要用下一个字节来继续表示该整数的剩余部分（需要时将该比特设为 **1**）

也就是说，这种编码方式可以根据字节数的多少，分别表示如下的整数范围。

- **1** 字节：$\mathbf{0 \sim 2^7 - 1}$（用 **7** 个比特来存储数据）
- **2** 字节：$\mathbf{0 \sim 2^{14} - 1}$（用 **14** 个比特来存储数据）
- **3** 字节：$\mathbf{0 \sim 2^{21} - 1}$（用 **21** 个比特来存储数据）

例如，10 的 variable-byte 编码是“00001010”。

而 1030 的 variable-byte 编码是“10000100:00000110”（为了便于查看，我们在两个字节间插入了一个“:”）。

与以字节为单位进行压缩的 variable-byte 编码相比，由于 delta 编码和 gamma 编码都是在比特级别上进行编码的编码方式，所以可以达到更好的压缩率。但是，在解码时，由于必须进行位操作，所以后者的处理速度可能不如使用 variable-byte 编码时快。因此，为了加快检索处理的速度，也有很多项目会在压缩倒排文件时采用 variable-byte 编码。

由于以上这些编码方式本质上都是通过相同的策略对整数进行编码的，所以统称为无参数的编码方式（Parameterless Codes)。

与此相对，还有另一类需要引入参数，并根据参数的值改变其行为的编码方式。下面，我们就来介绍一下这种编码的代表——Golomb 编码。

▍Golomb 编码

使用 Golomb 编码对整数 x（$x \geq 0$）进行编码时，要使用参数 m （≥ 1）将 x 分解为如下形式。

$$x = m \times q + r$$

其中，q 为 x 除以 m 的商，r 为余数。接下来需要对 q 进行 unary 编码，对 r 进行如下所示的编码，最后再将这两个编码拼接在一起即可得到 Golomb 编码。在这个过程中，令 $b = \log_2 m$（向上取整）， $t = 2^b - m$。

- 当 $\mathbf{0 \leq r < t}$ 时，用比特宽度为 $\mathbf{b - 1}$ 的二进制编码表示 $\mathbf{r}$

- 当 $\mathbf{t \leq r < m}$ 时，用比特宽度为 $\mathbf{b}$ 的二进制编码表示 $\mathbf{r + t}$

下面以 9 为例，当 $m = 5$ 时，由于 $9 = 5 \times 1 + 4$（$q = 1$、$r = 4$、$m = 5$)，所以可以得出

$$b = \log_2 m = \log_2 5 \approx 3$$

$t = 2^b - m = 2^3 - 5 = 3\ (< r)$

因此，我们要对 q（1）进行 unary 编码，对 r（4）＋ t（3）进行比特宽度为 b（3）的二进制编码，最后将二者拼接起来后就得到了 9 的 Golomb 编码“10:111”。

另外，当 $m = 1$ 时，Golomb 编码等同于 unary 编码，当 $m = 2^k$(k=1, 2, 3…) 时，Golomb 编码等同于一种叫作 rice 的编码。也就是说，unary 编码和 rice 编码都是 Golomb 编码的特殊形式。

由于我们在 wiser 中实现的是 Golomb 编码，所以在后面的章节还会再进一步讲解它。

压缩的原理

倒排文件为什么能被压缩呢？

下面我们通过举例来说明。请试着考虑一下如何用二进制表示“1、2、3、4”这 4 个数字。由于只有 4 个数字，所以用 2 个比特应该就可以表示了。如下所示，在这里我们为每个数字都分配了等宽的比特序列。

1:00

2:01

3:10

4:11

此时，假设有 [1，3，1，1] 这样一个整数序列，若使用上面的比特序列表示该整数数列，就需要 2（比特）× 4（个）＝ 8 比特。

下面，假设我们知道了在该整数序列中，出现 1 的概率要比出现其他数字的概率高。那么，更聪明的做法就是将更短的比特序列分配给 1。

于是，基于这种想法，我们改变了比特序列的分配方式，新的分配结果如下所示。此时，为了判断比特序列中数字间的边界，我们需要为 2、3、4 分配 3 个比特，这一点还请诸位注意。

1:0

2:100

3:101

4:110

基于这种分配方式，对于同样的整数序列 [1，3，1，1]，用新分配的比特序列表示的话，就只需要

1（比特）**×3**（个）＋**3**（比特）**×1**（个）＝**6**（比特）

与之前相比减少了 2 个比特。

实际上，压缩倒排文件时应用的也是与此相同的原理。也就是说，之前所讲解的各种编码方法，都是在预测出“倒排列表中的整数序列是以哪种分布方式排列的”之后，再基于这种预测，尝试对整数序列进行压缩的。

例如，对于整数 x 而言，由于 gamma 编码中的 unary 编码部分占用 $1+\log x$ 个比特，二进制编码部分占用 $\log x$ 个比特，所以表示 x 需要 $1+2\log x$ 个比特。根据香农（Claude Elwood Shannon）提出的信息量的定义[2]并基于整数 x 的编码长度，我们求得了如下所示的整数 x 的出现概率。

> [2]香农对信息量作出了如下定义，当某个事件 E 发生的概率为 p 时，该事件所含有的信息量为 $-\log_2 p$。虽然不够严谨，但是可以这样通俗地解释一下：当有 n 个事件将要发生时，这些事件可以用 $\log_2 n=-\log_2 1/n$ 表示。由于其中每个事件发生的概率都为 $1/n$，所以把这个概率设为 p，就可以用 $-\log_2 p$ 表示了。

$$l_x = -\log Pr[x]$$

$$Pr[x] = 2^{-l_x}$$

$$Pr_{\text{gamma}}[x] \approx 2^{-(1+2\log x)}$$

$$= 1/2x^2$$

这是 gamma 编码默认具备的概率分布。也就是说，在 gamma 编码中，我们是以整数 x 会以 $1/2x^2$ 的概率出现为前提进行编码的。整数序列中的预期概率分布与实际整数序列的分布越接近，编码方法所能达到的压缩率也就越高。

简而言之，人们已经证明出，在使用 gamma 编码等称为 Universal 编码的编码方式对整数序列进行压缩时，无论整数序列的分布如何，编码的长度都是最佳编码长度的常数倍。而且，在压缩由文档编号的差值构成的整数序列时，由于文档编号的差值序列遵从几何分布，所以通过使用在概率分布上拥有几何分布的 Golomb 编码进行编码，可以达到更高的压缩率。

5-2　实现wiser 中的压缩功能

压缩功能源代码的概要

下面，我们再来看一下在 wiser 中实现了压缩功能的那一部分源代码。另外，本书在讲解的过程中，会将压缩称为编码，将解压缩称为解码。

在 wiser 中，我们通过 Golomb 编码实现了编码和解码，并通过在函数 update_postings() 中调用的函数 encode_postings() 和 decode_postings() 实现了倒排列表的编码和解码。下面就让我们分别看一下这两个函数的源代码。

```
/**
 * 对倒排列表进行转换或编码
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] postings 转换或编码前的倒排列表
 * @param[in] postings_len 转换或编码前的倒排列表中的元素数
 * @param[out] postings_e 转换或编码后的倒排列表
 * @retval 0 成功
 */
static int
encode_postings(const wiser_env *env,
                const postings_list *postings, const int postings_len,
                buffer *postings_e)
{
  switch (env->compress) {
  case compress_none:
    return encode_postings_none(postings, postings_len, postings_e);
  case compress_golomb:
    return encode_postings_golomb(db_get_document_count(env),
                                  postings, postings_len, postings_e);
  default:
    abort();
  }
}

/**
 * 对倒排列表进行还原或解码
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] postings_e 待还原或解码的倒排列表
 * @param[in] postings_e_size 待还原或解码的倒排列表的字节数
 * @param[out] postings 还原或解码后的倒排列表
```

```
 * @param[out] postings_len 还原或解码后的倒排列表中的元素数
 * @retval 0 成功
 */
static int
decode_postings(const wiser_env *env,
                const char *postings_e, int postings_e_size,
                postings_list **postings, int *postings_len)
{
  switch (env->compress) {
  case compress_none:
    return decode_postings_none(postings_e, postings_e_size,
                                postings, postings_len);
  case compress_golomb:
    return decode_postings_golomb(postings_e, postings_e_size,
                                  postings, postings_len);
  default:
    abort();
  }
}
```

这两个函数会根据是否要进行编码的标志（env->compress）去调用相应的编码、解码函数。无需对倒排列表进行编码时，调用的是下面这两个函数。

- **encode_postings_none()**
- **decode_postings_none()**

需要通过 Golomb 编码进行编码、解码时，调用的是下面这两个函数。

- **encode_postings_golomb()**
- **decode_postings_golomb()**

了解无需进行压缩时的操作

在了解如何通过 Golomb 编码进行编码处理之前，让我们先来看一下在无需编码时被调用的函数 encode_postings_none() 和 decode_postings_none()。

函数 encode_postings_none() 的作用是将倒排列表转换成字节序列。也

就是说，该函数会先从倒排列表的各元素中取出文档编号、位置信息的数量以及位置信息的数组，然后再将这些数据以二进制的形式写入缓冲区。

```
/**
 * 将倒排列表转换成字节序列
 * @param[in] postings 倒排列表
 * @param[in] postings_len 倒排列表中的元素数
 * @param[out] postings_e 转换后的倒排列表
 * @retval 0 成功
 */
static int
encode_postings_none(const postings_list *postings,
                     const int postings_len,
                     buffer *postings_e)
{
  const postings_list *p;
  LL_FOREACH(postings, p) { ❶
    int *pos = NULL;
    append_buffer(postings_e, (void *)&p->document_id, sizeof(int)); ❷
    append_buffer(postings_e, (void *)&p->positions_count, sizeof(int)); ❸
    while ((pos = (int *)utarray_next(p->positions, pos))) { ❹
      append_buffer(postings_e, (void *)pos, sizeof(int)); ❺
    }
  }
  return 0;
}
```

首先，我们通过调用 utarray 的宏 LL_FOREACH()，从倒排列表中逐一取出各个文档编号的出现位置信息（❶）。

然后，将文档编号和存放出现位置信息的数组的大小（出现位置的数量）分别添加到缓冲区中（❷、❸），接着将各个出现位置（❹）也添加到该缓冲区中（❺）。

而在函数 decode_postings_none() 中进行的是函数 encode_postings_none() 的逆操作。即依次取出文档编号、位置信息的数量以及各个位置信息。

```
/**
 * 将字节序列的倒排列表还原为倒排列表
 * @param[in] postings_e 待还原的倒排列表（字节序列）
```

```
 * @param[in] postings_e_size 待还原的倒排列表（字节序列）中的元素数
 * @param[out] postings 还原后的倒排列表
 * @param[out] postings_len 还原后的倒排列表中的元素数
 * @retval 0 成功
 *
 */
static int
decode_postings_none(const char *postings_e, int postings_e_size,
                     postings_list **postings, int *postings_len)
{
  const int *p, *pend;

  *postings = NULL;
  *postings_len = 0;
  for (p = (const int *)postings_e,
       pend = (const int *)(postings_e + postings_e_size); p < pend;) {
    postings_list *pl;
    int document_id, positions_count;

    document_id = *(p++);
    positions_count = *(p++);
    if ((pl = malloc(sizeof(postings_list)))) {
      int i;
      pl->document_id = document_id;
      pl->positions_count = positions_count;
      utarray_new(pl->positions, &ut_int_icd);
      LL_APPEND(*postings, pl);
      (*postings_len)++;

      /* decode positions */
      for (i = 0; i < positions_count; i++) {
        utarray_push_back(pl->positions, p);
        p++;
      }
    } else {
      p += positions_count;
    }
  }
  return 0;
}
```

抓住 **Golomb** 编码的要点

无需对倒排列表进行编码时，我们只需将被直接转换成了字节序列的倒排列表写入数据库即可。而需要对倒排列表进行编码时，就要先对以下

3 种取自倒排列表的信息进行编码，然后再将转换成字节序列后的数据写入数据库。

- 已存储的文档编号
- 位置信息的数量
- 位置信息的数组

仅看源代码的话，也许还是难以轻松理解 Golomb 编码的过程，那么我们就来看一个具体的示例，边看边理解在 Golomb 编码的过程中都会进行怎样的处理吧。

假设有 [13，22，23，40] 这样一个文档编号的序列，我们试着考虑一下应该如何通过 Golomb 编码对其进行编码。首先要计算出这个整数序列中的所有后项与前项的差值。而对于第一个数字，则要计算它和 0 的差值。于是就得到了 [13，9，1，17] 这样一个整数序列。接下来，将所有的数字都减去 1。因为这样可以利用上文档编号序列中没有重复的编号，并且所有的前后项之差都是大于 1 的数值这个特点，使整数序列可以用更小的数值表示。于是，我们又得到了 [12，8，0，16] 这样一个整数序列。接下来，就开始用 Golomb 编码对这个整数序列进行编码。

正如前文所述，进行 Golomb 编码时要使用参数 m。我们都知道，对于参数 m，将它的值设为待编码的整数序列的平均值可以得到较高的压缩率。因此，在本例中我们将 m 的值设为该整数序列的平均值 9。

如下所示，Golomb 编码使用了参数 m 以及 q 和 r 两个整数来表示一个整数 n。整数 b 和 t 用于对整数 r 进行编码，其取值由 m 决定（ $b = \log_2 m$ 向上取整，$t = 2^b - m$）。当 $m = 9$ 时，$b = 4$、$t = 7$。

$$n = q \times m + r$$

▍用 Golomb 编码对整数序列进行编码

下面就让我们开始用 Golomb 编码对整数序列 [12，8，0，16] 进行编码吧。

- 对第 **1** 个整数进行编码

首先，从整数序列中取出第一个整数 12。然后用 12 除以 m 并舍去小数部分，得出了 q 的值为 1。

接下来开始编码，首先我们用 unary 编码来表示 q 的值。unary 编码是一种通过将 1 个 0 附加到 n 个连续的 1 之后来表示数值 n 的编码方式。也就是说，用 unary 编码表示 1 的话，得到的比特序列是 10。

接下来，计算 12 除以 m 的余数，又得到了 r 的值为 3。

表示 r 时要从以下 2 种模式中选择一种。

- 当 $\boldsymbol{r}$ 小于 $\boldsymbol{t}$ 时

  →用比特宽度为 b - 1 的二进制比特序列表示 r 的值

- 当 $\boldsymbol{r}$ 不小于 $\boldsymbol{t}$ 时

  →用比特宽度为 b 的二进制比特序列表示 $r + t$ 的值

由于此时 $r < t$（3 ＜ 7），所以要用 3 位二进制数表示 3，再用得到的比特序列“011”来表示 r 的值。

至此为止，我们得到的比特序列为“10011”。

- 对第 **2** 个整数进行编码

  下面，从整数序列取出第二个整数 8。此时 $q = 0$，$r = 8$。用 unary 编码表示 q，得到的比特序列为“0”。由于 $r \geq t$（8 ＞ 7），所以要用 4 个比特的比特序列“1111”来表示 $r + t$。

  至此为止，我们得到的比特序列为“10011011:11”。为了便于阅读，我们在每 8 个比特之后都插入了一个“:”，作为字节间的边界（余同)。

- 对第 **3** 个整数进行编码

  接下来我们取出了第 3 个整数 0，并按照之前的方法，求出了 q 和 r 的值。此时 $q = 0$，$r = 0$。用 unary 编码表示 q，得到的比特序

列为 0。由于 $r < t$（0 ＜ 7），所以要用由 3 个比特的比特序列“000”来表示 r 的值。

至此为止，我们得到的比特序列为“10011011:110000”。

- 对第 **4** 个整数进行编码

  最后，我们取出了最后一个整数 16。此时，$q = 1$，$r = 7$。用 unary 编码表示 q，得到的序列为 10。由于 $r \geq t$（7 ＝ 7），所以要用由 4 个比 特表示的比特序列“1110”来表示 $r + t$。

  最终我们得到的比特序列为“10011011:11000010:1110”。

  至此，对该整数序列的 Golomb 编码就结束了。可以看出，最终只需要2.5 个字节即可表示经过排序的整数序列 [13，22，23，40]。

  在存储这 2.5 个字节的比特序列时，为了以 1 个字节（＝ 8 个比特）为最小单位，我们在最后填充了 4 个值为 0 的比特，填充后的比特序列为“10011011:11000010:11100000”。

▌将比特序列解码成原先的整数序列

下面，再让我们试试能否将刚刚经过编码的比特序列解码为原先的整数序列吧。解码时要进行与编码时相反的操作。

要想解码，必须事先知道经过编码的整数个数以及参数 m 的取值。在这里，我们已知比特序列中存储了 4 个整数，并且参数 m 的值是 9。

- 对第 **1** 个整数进行解码

  首先，要从比特序列的起始位置开始查找值为 0 的比特。第 2 个比特的值就是 0。如果第 n 个比特的值为 0，那么就将从查找的起始位置到第 n - 1 个比特之间的数值作为 q，用 unary 编码进行解码。

  然后从刚找到的值为 0 的比特开始，再向后读取 b - 1 个（3 个）比特，得到比特序列“011”。将用二进制表示的 011 转换为十进制的整 数后，得到的结果是 3。由于求得的整数 3 小于 t（7），所以就直接将求得的数值作为 r 的值。因此，$r = 3$。

既然已经知道了 3 个参数 m、q 和 r 的取 值，那么就可以算出 1×9 ＋ 3 ＝ 12。由此就正确地解出了第一个整数 12。

至此，我们就得到了 [12] 这样一个整数序列。

- **对第 2 个整数进行解码**

  接下来继续寻找值为 0 的比特。正好第一个比特就是 0。因此 q ＝ 0。

  然后继续向后读取 b - 1 个（3 个）比特，得到了比特序列“111”。二进制的“111”即十进制的 7。

  由于求得的数值 7 不小于 t（7），所以还要再读取 1 个比特。即最终读取到的比特序列是“1111”。二进制的“1111”即十进制的 15。接下来只需再减去 t 即可求得 r 的值，即 r ＝ 15 - t ＝ 15 - 7 ＝ 8。

  根据参数 m、q 和 r，即可解出第二个整数，即 0×9 ＋ 8 ＝ 8。

  至此，我们得到的数列是 [12，8]。

- **对第 3 个和第 4 个整数进行解码**

  只需进行同样的运算（请务必亲自计算一下），即可进一步解出 0 和 16 两个整数。

  由于已知存储了 4 个整数，所以解码过程可以就此结束。

  我们最终得到了整数序列 [12，8，0，16]。这说明经过 Golomb 编码后的比特序列能够被正确地解码。

- **加上前后项差值后再将各个整数加 1**

  由于这个整数序列中的每个整数（除第一个整数以外）其实都是与前 1 个整数的差值，所以我们还要将这个差值加到每个整数上。然后还要再对所有整数都加上 1。至此我们就得到了原先的整数数列 [13，22，23，40]。

解读 **Golomb** 编码中的编码处理

既然已经了解了 Golomb 编码的概念，下面我们就来实际了解一下函数 encode_postings_golomb()。

```
/**
 * 对倒排列表进行Golomb编码
 * @param[in] documents_count 文档的总数
 * @param[in] postings 编码前的倒排列表
 * @param[in] postings_len 编码前倒排列表中的元素个数
 * @param[out] postings_e 编码后的倒排列表
 * @retval 0 成功
 */
static int
encode_postings_golomb(int documents_count,
                       const postings_list *postings, const int postings_le
                       buffer *postings_e)
{
  const postings_list *p;

  append_buffer(postings_e, &postings_len, sizeof(int)); ❻
  if (postings && postings_len) {
    int m, b, t;
    m = documents_count / postings_len; ❼
    append_buffer(postings_e, &m, sizeof(int)); ❽
    calc_golomb_params(m, &b, &t); ❾
    {
      int pre_document_id = 0;

      LL_FOREACH(postings, p) { ❿
        int gap = p->document_id - pre_document_id - 1; ⓫
        golomb_encoding(m, b, t, gap, postings_e); ⓬
        pre_document_id = p->document_id;
      }
    }
    append_buffer(postings_e, NULL, 0); ⓭
  }
  LL_FOREACH(postings, p) { ⓮
    append_buffer(postings_e, &p->positions_count, sizeof(int));
    if (p->positions && p->positions_count) {
      const int *pp;
      int mp, bp, tp, pre_position = -1;

      pp = (const int *)utarray_back(p->positions); ⓯
      mp = (*pp + 1) / p->positions_count; ⓰
      calc_golomb_params(mp, &bp, &tp); ⓱
      append_buffer(postings_e, &mp, sizeof(int));
      pp = NULL;
      while ((pp = (const int *)utarray_next(p->positions, pp))) {
```

```
        int gap = *pp - pre_position - 1;
        golomb_encoding(mp, bp, tp, gap, postings_e);
        pre_position = *pp;
      }
      append_buffer(postings_e, NULL, 0);
    }
  }
  return 0;
}
```

在 Gdomb 编码中，由于要使用整数序列中前后项的差值，所以我们要对文档编号和出现位置分别进行编码。

首先，将倒排列表中包含的文档数存储起来（❻）。

接下来，计算出用于对文档编号的差值序列进行 Golomb 编码的参数 *m* 的取值（❼）。在这里，我们使用了文档编号差值的平均值作为参数 *m* 的取值。文档编号差值的平均值可以使用文档总数除以倒排列表中包含的文档数计算得出。由于在解码时还要用到参数 *m*，所以在这里我们还要先将参数 *m* 存储起来（❽）。

接下来，计算出由参数 *m* 唯一决定的参数 *b* 和 *t* 的取值（$b = \log_2 m$ 向上取整，$t = 2^{b-m}$）（❾）。

然后逐一取出倒排列表中的文档编号（❿）。每取出一个文档编号，就计算其与刚刚存储的文档编号的差值（⓫）。随后用函数 golomb_encoding() 对计算结果进行编码（⓬），最后以比特为单位将编码结果添加到变量 postings_e 中。我们先继续往下看，稍后再讲解函数 golomb_encoding() 的实现过程。

接下来，再次通过函数 append_buffer()，将以比特为单位的信息统一为以字节为最小单位的信息（⓭）。

在标有⓮的循环中，又对由词元先后出现位置的差值构成的整数序列进行了编码。虽然基本的流程与对文档编号进行编码的流程大致相同，但是由于词元在每个文档中的出现次数都不相同，所以要对每个文档单独计算 Golomb 编码中的参数 *m*。

在 ⓯ 的步骤 中，通过调用能返回数组中最后一个元素的函数

utarray_back()，获取了词元在文档中最后一次出现的位置。

接下来，用词元在文档中最后一次出现的位置除以词元在文档中的出现次数，计算出了对出现位置进行编码时需要用到的参数 *m*（变量 mp）（⓰）。

在⓱的步骤后，我们又使用了与之前从❾到⓭对文档编号进行编码时相同的流程，对出现位置数组中的整数进行了编码。

函数 **golomb_encoding()**

下面，我们来看一下负责实际编码工作的函数 golomb_encoding()。

```
/**
 * 用Golomb编码对1个数值进行编码
 * @param[in] m Golomb编码中的参数m
 * @param[in] b Golomb编码中的参数b。ceil(log2(m))
 * @param[in] t pow2(b) - m
 * @param[in] n 待编码前的数值
 * @param[out] buf 编码后的数据
 */
static inline void
golomb_encoding(int m, int b, int t, int n, buffer *buf)
{
  int i;
  /* encode (n / m) with unary code */
  for (i = n / m; i; i--) { append_buffer_bit(buf, 1); } ⓲
  append_buffer_bit(buf, 0); ⓳
  /* encode (n % m) */
  if (m > 1) { ⓴
    int r = n % m;
    if (r < t) {
      for (i = 1 << (b - 2); i; i >>= 1) {
        append_buffer_bit(buf, r & i);
      }
    } else {
      r += t;
      for (i = 1 << (b - 1); i; i >>= 1) {
        append_buffer_bit(buf, r & i);
      }
    }
  }
}
```

首先，通过 unary 编码对 *n* / *m* 的结果进行编码（⓲），然后通过函数 append_buffer_bit() 将编码后的比特序列写入到缓冲区中（⓳）。

接下来，从⓴的步骤开始，对由 *n* % *m* 计算出的数值 *r* 进行了编码，并将结果也写入到了缓冲区中。

解读 **Golomb** 编码的解码处理

下面，让我们再来看一下函数 decode_postings_golomb()。该函数的作用是将经过 Golomb 编码后的二进制序列解码为倒排列表。

```
/**
 * 对经过Golomb编码后的倒排列表进行解码
 * @param[in] postings_e 经过Golomb编码后的倒排列表
 * @param[in] postings_e_size 经过Golomb编码后的倒排列表中的元素数
 * @param[out] postings 解码后的倒排列表
 * @param[out] postings_len 解码后的倒排列表中的元素数
 * @retval 0 成功
 */
static int
decode_postings_golomb(const char *postings_e, int postings_e_size,
                       postings_list **postings, int *postings_len)
{
  const char *pend;
  unsigned char bit;

  pend = postings_e + postings_e_size;
  bit = 0x80; ㉑
  *postings = NULL;
  *postings_len = 0;
  {
    int i, docs_count;
    postings_list *pl;
    {
      int m, b, t, pre_document_id = 0;

      docs_count = *((int *)postings_e); ㉒
      postings_e += sizeof(int);
      m = *((int *)postings_e); ㉓
      postings_e += sizeof(int);
      calc_golomb_params(m, &b, &t);
      for (i = 0; i < docs_count; i++) { ㉔
        int gap = golomb_decoding(m, b, t, &postings_e, pend, &bit); ㉕
        if ((pl = malloc(sizeof(postings_list)))) {
```

```
        pl->document_id = pre_document_id + gap + 1; ㉖
        utarray_new(pl->positions, &ut_int_icd);
        LL_APPEND(*postings, pl);
        (*postings_len)++;
        pre_document_id = pl->document_id;
      } else {
        print_error("memory allocation failed.");
      }
    }
  }
  if (bit != 0x80) { postings_e++; bit = 0x80; } ㉗
  for (i = 0, pl = *postings; i < docs_count; i++, pl = pl->next) {
    int j, mp, bp, tp, position = -1;

    pl->positions_count = *((int *)postings_e);
    postings_e += sizeof(int);
    mp = *((int *)postings_e);
    postings_e += sizeof(int);
    calc_golomb_params(mp, &bp, &tp);
    for (j = 0; j < pl->positions_count; j++) {
      int gap = golomb_decoding(mp, bp, tp, &postings_e, pend, &bit);
      position += gap + 1;
      utarray_push_back(pl->positions, &position);
    }
    if (bit != 0x80) { postings_e++; bit = 0x80; }
  }
 }
 return 0;
}
```

首先，我们要对文档编号的整数序列进行解码。

为此，需要先初始化表示当前正指向二进制序列中哪个比特的变量 bit（㉑）。0x80 表示当前正指向第 0 个比特。有关该变量的细节，将在稍后讲解函数 golomb_decoding() 时再一同讲解。

接下来，在㉒和㉓的步骤中，分别读取了文档数（变量 docs_count）和用Golomb 编码进行压缩时所要用到的参数 *m*。

然后对于每个文档（㉔），通过调用函数 golomb_decoding()，对作为文档编号差值的整数（㉕）进行了解码。随后，又根据该差值，还原了原始的文档编号（㉖）。

最后，在对出现位置进行解码时（㉗之后），采用了与之前（从㉑到㉖）对文档编号进行解码时同样的处理过程。

▍函数 golomb_decoding()

下面，我们再来看一下函数 golomb_decoding()。其中的解码处理与在函数 golomb_encoding() 中进行的编码处理是相互对应的。

```
/**
 * 用Golomb编码解码一个数值
 * @param[in] m Golomb编码中的参数m
 * @param[in] b Golomb编码中的参数b。ceil(log2(m))
 * @param[in] t pow2(b) - m
 * @param[in] buf 作为解码对象的数据
 * @param[in] buf_end 作为解码对象的数据的末尾位置
 * @param[in] bit 作为解码对象的数据中的起始比特
 * @return 解码后的值
 */
static inline int
golomb_decoding(int m, int b, int t,
                const char **buf, const char *buf_end, unsigned char *bit)
{
  int n = 0;

  /* decode (n / m) with unary code */
  while (read_bit(buf, buf_end, bit) == 1) { ㉘
    n += m; ㉙
  }
  /* decode (n % m) */
  if (m > 1) { ㉚
    int i, r = 0;
    for (i = 0; i < b - 1; i++) { ㉛
      int z = read_bit(buf, buf_end, bit); ㉜
      if (z == -1) { print_error("invalid golomb code"); break; }
      r = (r << 1) | z; ㉝
    }
    if (r >= t) { ㉞
      int z = read_bit(buf, buf_end, bit); ㉟
      if (z == -1) {
        print_error("invalid golomb code");
      } else {
        r = (r << 1) | z; ㊱
        r -= t; ㊲
      }
    }
```

```
    n += r; ㊳
  }
  return n;
}
```

正如前文所述，变量 bit 的作用是以二进制数的形式表示要从变量 buf 的哪个位置开始读取。例如，当 bit 的值为 0x80 时，由于将其转换成二进制后左数第 1 个位置上的比特是 1，所以就表示要从 buf 的第 1 个比特开始读取。以此类推，当 bit 的值为 0x40 时，由于将其转换成二进制后左数第 2 个位置上的比特是 1，所以表示要从 buf 的第 2 个比特开始读取。之所以要用二进制数表示变量 bit，是因为只需要对 bit 和 buf 进行逻辑与运算，就可以从 buf 与 bit 中值为 1 的比特相对应的位置上读取出 1 个比特值。

下表列出了 bit 的各种取值分别能够读取 buf 中哪个位置上的比特值。

bit：**0x80** 二进制表示：**10000000** 指向的比特：**buf** 中的第**1** 个比特

bit：**0x40** 二进制表示：**01000000** 指向的比特：**buf** 中的第**2** 个比特

bit：**0x20** 二进制表示：**00100000** 指向的比特：**buf** 中的第**3** 个比特

bit：**0x10** 二进制表示：**00010000** 指向的比特：**buf** 中的第**4** 个比特

bit：**0x08** 二进制表示：**00001000** 指向的比特：**buf** 中的第**5** 个比特

bit：**0x04** 二进制表示：**00000100** 指向的比特：**buf** 中的第**6** 个比特

bit：**0x02** 二进制表示：**00000010** 指向的比特：**buf** 中的第**7** 个比特

bit：**0x01** 二进制表示：**00000001** 指向的比特：**buf** 中的第**8** 个比特

首先，我们通过 unary 编码对二进制序列进行了解码（㉘、㉙），二进制序列指的是在函数 golomb_encoding() 中进行⓲和⓳两处编码后得到的序列。正如前文所述，函数 read_bit()（㉘）会利用 buf 和 bit，从 buf 中读取出 1 个比特。根据 Golomb 编码的规则，只要当前 bit 所指向的 buf 中的比特值为 1，就要将 m 的值累加到 n 上，最后我们就通过这种方法，将“解码后的数值 ×m”的结果赋值给了变量 n（㉙）。

㉚之后的内容，是对二进制序列进行的解码处理，这里的二进制序列指的是在函数 golomb_encoding() 中进行⓴之后的编码所形成的序列。接下来在从㉛到㉝的步骤中，我们将二进制序列中前 b - 1 个比特的数据解码后赋值给了变量 r。具体做法是反复执行 b - 1 次㉛以后的操作：先从二进制序列中读取 1 个比特（㉜），然后将 r 左移 1 个比特，用刚刚读取出的 1 个比特值作为左移后空出的最低比特上的值（㉝）。

当 r 不小于 t 时（㉞），我们还要继续向后读取 1 个比特（㉟），然后只要再重复执行一遍㉝的操作（㊱），并减去 t 的值（㊲），即可解出 r 的值。而当 r 小于 t 时，由于此时 r 中存放的就是解码后的值，所以在累加到变量 n 之前就不需要再进行什么处理了。

最后只需要再将 n 和 r 加到一起，即可解出原来的数值了（㊳）。

至此为止，我们就了解了有关倒排列表的编码和解码处理了。如果感到比特操作不是很好理解，那么在准确地理解了 Golomb 编码之后，不妨试着一步步地梳理各个操作。

第 6 章　挑战**wiser**的优化及参数的调整

至此为止，我们已经从源代码级别介绍了全文搜索系统 wiser 的结构，诸位也体验到了搜索引擎的开发过程。wiser 是一个简单的搜索引擎，旨在帮助诸位理解搜索引擎的核心部分。因此，要想使之成为一个实用的搜索引擎，还需要大量的优化工作。另外，虽然 wiser 带有多个参数，但是我们还没有分析调整这些参数所能引起的此消彼长的变化。于是，我们想在本章以练习的形式来优化 wiser，确认调整 wiser 的参数后所能引起的各种此消彼长的变化，以此来帮助诸位理解构建实用搜索引擎的要点，从而加深诸位对搜索引擎结构及特性的认识。

对于练习的问题，我们会结合实际予以举例解答。其中有从正面解答问题的例子，也有回避了细枝末节的例子。不过，在翻阅答案前，请诸位一定要自己努力动手优化一下源代码。

6-1　提高检索处理的效率

在第 4 章的开始部分，我们讲解过 wiser 现有的检索处理方法还不够高效。既然如此，下面就让我们从这里开始优化吧。

优化检索处理

在 wiser 现有的检索处理过程中，我们采用了与构建倒排索引时同样的分割方法，即每次向后错开 1 个字符，将查询分割成了 bi-gram 的词元序列，并检查了分割出来的词元在文档中是否是按顺序排列的。在这个过程中有些地方是可以优化的。

下面我们就举例说明。假设我们用 bi-gram 的词元为文档“自制搜索引擎”构建出了如下的倒排索引。倒排索引中所有的文档编号均为 0。

- 自制**:0;0**
- 制搜**:0;1**
- 搜索**:0;2**
- 索引**:0;3**
- 引擎**:0;4**

当通过查询字符串“自制搜索引擎”检索上述倒排索引时，只要将其分割成“自制”“搜索”“引擎”3 个词元，并观察到在文档中后一个词元的出现位置（偏移量）总是与前一个词元的出现位置相距 2 个字符，就可以断定短语“自制搜索引擎”出现在了文档中。也就是说，对于由每次向后错开 1 个字符而形成的 bi-gram 的词元构建的倒排索引而言，只需要将查询分割为若干个无重复部分的词元序列即可。

通过这样的分割，就可以减少需要参与检索的词元的数量。这也就意味着这样的分割有助于减少关联到词元上的倒排列表的获取次数，从而降低对多个倒排列表中的出现位置信息进行相邻判定时的比较处理的次数。在获取倒排列表时通常都会伴随着大量的 I/O 操作，而进行比较处理时又通常会消耗大量的 CPU 资源，因此只要减少了词元的数量，就

能大幅度地提升检索处理的效率。

将查询分割为无重复部分的词元序列

下面，我们来看一下如何将查询分割为无重复部分的词元序列。在这里，我们需要对函数 text_to_postings_lists() 进行改造，该函数会被从字符串中提取词元的函数 split_query_to_tokens() 所调用。

使用 N-gram 进行分割时，改造后的常规做法是从查询字符串的起始位置开始不断地取出一组组无重复部分的 *n* 个字符。但是，当查询字符串的长度不能被 *n* 整除时，就会在最后留下一个长度小于 *n* 的词元。遇到这种特殊情况时，我们要将末尾的 *n* 个字符作为 1 个词元。以“查询字符串”为例，我们先分别用 bi-gram 和 tri-gram 对其进行分割，分割结果如下所示。

- 使用 **bi-gram** 的分割结果：**“**查询**”“**字符**”“**符串**”**

- 使用 **tri-gram** 的分割结果：**“**查询字**”“**字符串**”**

像这样，当查询字符串的长度不能被 *n* 整除时，我们可以通过如下的策略，生成含有部分重复字符的词元。另外，我们将变量 position 用作游标，指向查询中正在处理的字符。

- 当 **position** 可以被 ***n*** 整除并且候选词元的长度不小于 ***n*** 时 → 将该候选词元作为正式词元

- 当 **position** 可以被 ***n*** 整除并且候选词元的长度小于 ***n*** 时 → 将该候选词元的前一个候选词元作为词元

由于函数 ngram_next() 是通过每次向后错开 1 个字符来获取词元的，所以可以保证位于最后一个长度小于 *n* 的词元之前的候选词元其长度一定是 *n*。因此，我们就需要定义 3 个新的变量 last_t_len、last_t 以及 last_position，用于存储前一个候选词元的信息。

```
int
text_to_postings_lists(wiser_env *env,
                       const int document_id, const UTF32Char *text,
                       const unsigned int text_len,
                       const int n, inverted_index_hash **postings)
```

```
{
  /* FIXME: now same document update is broken. */
  int t_len, position = 0;
  const UTF32Char *t = text, *text_end = text + text_len;
  int last_t_len = 0, last_position = 0;
  const UTF32Char *last_t = NULL;

  inverted_index_hash *buffer_postings = NULL;

  for (; (t_len = ngram_next(t, text_end, n, &t)); t++, position++) {
    int filtered_t_len = 0, filtered_position;
    const UTF32Char *filtered_t = NULL;

    /* 在检索时，基本上是当position可以被n整除时才取出词元 */
    if (document_id || ((position % n == 0) && t_len >= n)) {
        filtered_t_len = t_len;
        filtered_t = t;
        filtered_position = position;
    /* 但是，要保证最后一个词元含有n个字符 */
    } else if (t_len < n) {
      if (last_t_len && last_t) {
        filtered_t_len = last_t_len;
        filtered_t = last_t;
        filtered_position = last_position;
      } else {
          break;
      }
    }

    if (filtered_t_len && filtered_t) {
      int retval, t_8_size;
      char t_8[n * MAX_UTF8_SIZE];

      utf32toutf8(filtered_t, filtered_t_len, t_8, &t_8_size);

      retval = token_to_postings_list(env, document_id, t_8, t_8_size,
                                      filtered_position, &buffer_postings);
      if (retval) { return retval; }

      last_t_len = 0;
      last_t = NULL;
    } else {
      last_t_len = t_len;
      last_t = t;
      last_position = position;
    }
  }
```

```
  if (*postings) {
    merge_inverted_index(*postings, buffer_postings);
  } else {
    *postings = buffer_postings;
  }

  return 0;
}
```

分别用优化前和优化后的 wiser 检索同一个查询后可以发现，检索结果的数量并未发生变化，而检索处理的速度则有所提升。但是由于索引中收录的文档非常多，而且查询的长度又没有达到足够的长度，所以也许并不能切实感到检索处理的速度提升了。

另外，分别用优化前和优化后的 wiser 检索同一个查询后，还会发生得到了不同的检索结果的情况。例如，用优化后的 wiser 去检索查询字符串“漫画作品”所得到的结果，就比用优化前的 wiser 所得到的结果多[1]。这意味着在上述示例代码中有 Bug 吗？

> [1]优化后的 wiser 会将“漫画作品”分割为词元“漫画”和“作品”，而优化前的 wiser 会将其分割成“漫画”“画作”和“作品”。——译者注

其实不是这样的。之所以检索结果的数量不一致，是因为在 Wikipedia 的词条中有时会出现用形如“[[漫画]] 作品”的 Wiki 标记书写的内容。而在构建索引时，wiser 的词元分割器（Tokenizer）会把这样的标记视作空格，即等价于将内容为“漫画 作品”的字符串分割为词元序列[2]。所以用优化前的 wiser 检索“漫画作品”时，只要文档中没有出现“画作”这个词元，该文档就会从检索中遗漏。

> [2]分割的结果为“漫画”“画”“作品”“品”。——译者注

6-2 禁用短语检索

在 wiser 中，我们会将一个个二元组存储到倒排索引的倒排列表里，二元组中包括含有某个词元的文档编号和该词元出现的位置。这样的倒排列表称为单词级别的倒排列表。而且我们还讲解过，通过单词级别的倒排列表可以准确地找出包含在查询中的短语。

那么，如果不进行短语检索，会产生多少检索噪声呢？在 wiser 中，启动参数 -s 可禁用短语检索。下面我们就使用这个参数来亲身体验一下检索噪声吧。

分析对 2 字符的字符串进行检索时的行为

首先，让我们来检索一个 2 字符的字符串，例如“漫画”。在检索时，我们使用的是为 1000 个词条建立索引后形成的数据库。检索分两次进行，第一次不带参数 -s，第二次带上该参数。

```
> ./wiser -q '漫画' wikipedia_1000.db
...
Total 12 documents are found!
...

> ./wiser -q '漫画' -s wikipedia_1000.db
...
Total 12 documents are found!
...
```

可以发现两次的检索结果是一致的。所谓 bi-gram，就是指长度为 2 个字符的词元，因此对 bi-gram 进行短语检索的意义不大。

分析对 3 字符的字符串进行检索时的行为

下面，我们再来检索一个 3 字符的字符串。例如“第一个”。

```
> ./wiser -q '第一个' wikipedia_1000.db
...
document_id: 775 title: 10月 score: 3.244442
```

```
document_id: 511 title: 朝鲜 (称谓) score: 3.174791
document_id: 553 title: Wikipedia:互助客栈档案室 score: 3.174791
document_id: 664 title: A score: 3.174791
document_id: 708 title: 6月25日 score: 3.174791
document_id: 395 title: 武术 score: 2.139744
document_id: 765 title: 7月 score: 2.139744
document_id: 766 title: 5月 score: 2.139744
document_id: 806 title: 雷 score: 2.139744
document_id: 914 title: 星期二 score: 2.139744
Total 154 documents are found!
[time] 2015/10/14 05:28:07.000007 (diff   1.073670)
```

接下来，禁用短语检索后再检索一次“第一个”。

```
> ./wiser -q '第一个' -s wikipedia_1000.db
...
document_id: 392 title: 约翰·沃尔夫冈·冯·歌德 score: 16.082908
document_id: 260 title: 唐朝 score: 16.013257
document_id: 830 title: 围棋 score: 16.013257
document_id: 446 title: 联合国 score: 15.943607
...
document_id: 806 title: 雷 score: 2.139744
document_id: 883 title: 怀俄明州 score: 2.139744
document_id: 914 title: 星期二 score: 2.139744
document_id: 941 title: F# score: 2.139744
Total 349 documents are found!
[time] 2015/10/14 05:30:13.000006 (diff   1.030619)
```

可以看出，检索结果在数量上有较大的差距。禁用短语检索前能找到154条结果，而禁用短语检索后竟能找到349条结果。究其原因可以发现，无论“第一”还是“一个”都是会在大量文档中出现的词元。在诸如“世界上第一部（架、本……）……”“……一个时代（系统、周期……）”等句子中，就经常会遇到虽然出现了“第一”和“一个”，但是却没有出现“第一个”的情况。除此之外，请再试着比较一下在禁用短语检索前后，用“东西南北”“中国大学”等短语检索所得到的结果。可以发现，检索结果在数量上依然存在着较大的差距。因此，虽然禁用短语检索可以省略查找短语的过程，从而提高检索的速度，但是这样做会导致那些原本与查询并不相关的文档最终也出现在了检索结果中。

6-3　改变检索结果的输出顺序

作为检索结果排序核心的指标

检索系统有时会产生大量的检索结果。此时即使是将检索结果原封不动地返回，用户也无法查阅完所有内容。因此更好的做法是根据某种指标进行评分，然后只将得分较高的文档作为检索结果呈现给用户。

下面我们就来介绍几个用于对检索结果排序的指标（属性）。

❙ TF-IDF

TF 是 Term Frequency（词频）的缩写，用于描述特定词元在某个特定文档中的出现次数。IDF 是 Inverse Document Frequency（逆文档频率）的缩写，指在所有的文档中，出现过特定词元的文档数的倒数。TF-IDF 值就是上述 TF 和 IDF 的乘积。

假设某个词元在某个文档中的出现次数为 *T*，总共有 *A* 个文档，并且某个词元至少出现过 1 次的文档数为 *D*，那么 TF-IDF 值的计算公式如下所示：

$$\mathrm{tf} = T$$

$$\mathrm{idf} = \log \frac{A}{D}$$

$$\mathrm{tf} - \mathrm{idf} = \mathrm{tf} \times \mathrm{idf}$$

由公式可以看出，在同一个文档中，作为检索对象的词元出现的次数越多，其 TF 值就越高。而如果除了少数几个文档以外，某个词元在大多数文档中都没有出现，那么该词元的 IDF 值就会增大。也就是说，虽然有的词元没有在大多数文档中频繁出现，但是频繁出现在少数文档中同样会使该词元的 TF-IDF 值增大。因此，一般会把 TF-IDF 作为衡量“词元在文档集合中是否特殊”的一个指标。

在 TF-IDF 的计算方法之中有各种各样的变体。例如，当文章很长时，所有词元的 TF 都会变得过大，因此在有的计算方法中，就需要通过除以文章中出现次数最多的词元的 TF 来平滑数据。另外，由于 TF-IDF 值与 TF 值呈线性关系，即 TF 值增大 1 倍，作为得分的 TF-IDF 值也会

增大一倍，所以还有的计算方法会对 TF 取对数。

$$\mathrm{tf} = \begin{cases} 1+\log(T) & (T>0) \\ 0 & (T=0) \end{cases}$$

$$\mathrm{idf} = \log\frac{A}{D}$$

$$\mathrm{tf}-\mathrm{idf} = \mathrm{tf}\times\mathrm{idf}$$

❙ 文档的最后更新日期

对于某些作为检索对象的文档集合，搜索引擎会根据文档的最后更新日期，而不是查询与文档的相关度，对检索结果进行排序。

例如在 Twitter 的推文检索功能中，就是按照发布日期的降序来呈现检索到的推文的。之所以这样做，是因为用户想浏览的是“最近大家都在热议什么话题”。面对“用户想了解的是特定的新闻以及大家对这条新闻的评论”这种需求，将发布日期最新的新闻放到最上面再自然不过了。而且，由于 Twitter 的用户界面在设计之初就是按照时间顺序列出推文的，所以检索结果也沿用这种风格显示的话，浏览起来会更加方便。

❙ **PageRank**

在对检索结果排序时，Google 会计算一种名为 PageRank 的独有指标，并将其结果作为决定 Web 检索结果呈现顺序的要素之一。

PageRank 的计算，是基于“从受欢迎的网页中精挑细选出的链接所指向的网页应该也很受欢迎吧”这种假设。具体来说，是基于以下 3 个推测。

- 被很多网站的链接指向的网页从某种程度上来说是优质的、受欢迎的
- 被受欢迎网页上的链接指向的网页从某种程度上来说也是优质的
- 网页上的链接的数量与搜索引擎对链接目标网页的推荐程度呈反比

由于 PageRank 的计算只与网页间的链接结构有关，所以计算时会完全忽略网页的内容。在这一点上，PageRank 与 TF-IDF 等计算时只参考文档内容的指标完全不同。

现有的 wiser 在输出前会根据 TF-IDF 值的大小对检索结果进行降序排列，尽管如此，能够在输出前使用除此之外的排序方式也是一个不错的选择。为此，作为练习，下面就让我们改造一下 wiser，使其能够在输出前按照文档的大小对检索结果进行降序排列。

按照文档大小降序排列的检索结果

首先，我们需要创建一个能够获取文档大小的函数。幸运的是，SQLite 已经提供了函数 LENGTH()，用于获取列（Column）中存放的字符串的长度，因此我们可以直接利用这个函数。

为了添加新的数据库查询，首先需要预留一块空间以存储相应的准备语句（ Prepared Statements）。为此我们向 wiser.h 的结构体 struct_wiser_env 中添加了 1 个元素。

```
typedef struct _wiser_env {
  ...
  sqlite3_stmt *get_document_body_size_st;
  ...
} wiser_env;
```

然后在 database.c 中的函数 init_database() 中，创建一个准备语句。准备语句由用于获取文档大小的查询语句构成。

```
init_database(wiser_env *env, const char *db_path)
{
  ...
  sqlite3_prepare(env->db,
    "SELECT LENGTH(body) FROM documents WHERE id = ?;",
    -1, &env->get_document_body_size_st, NULL);
  ...
}

fin_database(wiser_env *env)
{
  ...
```

```
  sqlite3_finalize(env->get_document_body_size_st);
  ...
}
```

接下来，在 database.c 中创建一个用于获取文档大小的函数 db_get_document_size()。该函数的逻辑非常简单，就是以 document_id 为键获取文档的长度。

```
int
db_get_document_size(const wiser_env *env, int document_id,
                     const unsigned int *document_size)
{
  int rc;

  sqlite3_reset(env->get_document_body_size_st);
  sqlite3_bind_int(env->get_document_body_size_st, 1, document_id);

  rc = sqlite3_step(env->get_document_body_size_st);
  if (rc == SQLITE_ROW) {
    *document_size = (int)sqlite3_column_int(env->get_document_body_size_st
  }

  return 0;
}
```

实现了函数 db_get_document_size() 之后，我们还要在文件 database.h 中添加该函数的原型声明。

```
int db_get_document_size(const wiser_env *env, int document_id,
  const unsigned int *document_size);
```

进行到这一步，只要调用函数 db_get_document_size() 即可获取文档的大小了。

下面，我们还要将文档的大小存储到每个文档中，以使排序函数可以利用此数据对检索结果排序。为此，我们首先在 search.c 的结构体 search_results 中添加了一个用于存储文档大小的元素 body_size。

```
typedef struct {
```

```
  ...
  unsigned int body_size; /* 文档大小 */
  ...
}
```

然后通过调用函数 add_search_result()，从数据库中取出了成为检索结果的各文档的大小。为了访问数据库，我们还以参数的形式将 wiser_env 类型的变量 env 传递给了该函数。

```
/**
 * 将文档添加到检索结果中。
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] results 指向检索结果的指针
 * @param[in] document_id 要添加的文档编号
 * @param[in] score 得分
 */
static void
add_search_result(wiser_env *env, search_results **results, const int docum
                   const double score)
{
  search_results *r;
  if (*results) {
    HASH_FIND_INT(*results, &document_id, r);
  } else {
    r = NULL;
  }
  if (!r) {
    if ((r = malloc(sizeof(search_results)))) {
      r->document_id = document_id;
      r->score = 0;
      db_get_document_size(env, document_id, &r->body_size);
      HASH_ADD_INT(*results, document_id, r);
    }
  }
  if (r) {
    r->score += score;
  }
}
```

由于函数 add_search_result() 是由函数 search_docs() 调用的，所以要在函数 search_docs() 内部将变量 env 传递给函数 add_search_result()。

```
void
search_docs(wiser_env *env, search_results **results,
            query_token_hash *tokens)
{
...
        if (phrase_count) {
          double score = calc_tf_idf(tokens, cursors, n_tokens,
                                     env->indexed_count);
          add_search_result(env, results, doc_id, score);
        }
...
}
```

至此，我们终于能够从检索结果的结构体中取出文档大小了。

接下来，我们在 search.c 中创建根据 body_size 进行比较的比较函数 search_results_body_size_desc_sort()。

```
/**
 * 比较两条检索结果的文档大小
 * @param[in] a 一条检索结果
 * @param[in] b 另一条检索结果
 * @return 文档的大小关系
 */
static int
search_results_body_size_desc_sort(search_results *a, search_results *b)
{
  return (b->body_size > a->body_size) ? 1 :
         (b->body_size < a->body_size) ? -1 : 0;
}
```

创建好以后，只需要在函数 search_docs() 中用该函数取代先前调用的函数 search_results_score_desc_sort() 即可。

```
void
search_docs(wiser_env *env, search_results **results,
            query_token_hash *tokens)
{
  ...
  free_inverted_index(tokens);
  HASH_SORT(*results, search_results_body_size_desc_sort);
}
```

至此，我们终于完成了根据文档大小对检索结果排序的功能。该功能看起来好像很简单，但没想到实现起来竟然这么麻烦。

由于 Wikipedia 的 XML 中也包含了最后更新日期等信息，所以通过参考上述的修改过程进行改造，就应该能使 wiser 可以根据最后更新日期对检索结果排序了。下面就请诸位将此作为更进一步的需求，试着挑战一下吧。

专栏

排名欺诈

Web 检索系统的诞生使文档的排名具有了重要的经济价值，由此就导致了经常会发生通过做手脚来提升文档排名的行为。

假设有一个采用了 TF-IDF 来计算网页重要度的搜索引擎。那么，只要使网页中某个单词的出现次数翻倍，通过该单词进行检索时，相应网页的重要度就会随之翻倍。也就是说，只需要增加网页中的单词就可以轻松地操纵网页的重要度了。

为了防止有人通过上述简单易行的手段来操纵重要度，Google 采用了不依赖网页的内容，而是使用网页的链接结构来计算重要度的 PageRank。

但是，重要度归根到底只是一个根据某种标准计算出来的数值。所以经常会出现一些网页专家，通过推测搜索引擎使用的计算标准来牟取超过文档本身拥有价值的排名。从本质上来讲，这种相互较量很难避免，而Web 检索系统的运营者也经常会对此感到束手无策。

6-4　让1个字符的查询也能检索出结果

在 wiser 中，我们是用 bi-gram 来分割词元的。这就意味着不足 2 个字符的字符串是检索不出结果的。那么应该怎样调整才能使 1 个字符也能检索出结果呢？

由于此前我们已经将词元全部存储到了 SQLite 的 tokens 表中，所以现在只要像下面这样就能检索出以字符 X 开头的词元了。

WHERE token like 'X%'

只要先借助该方法将所有以某一个字符开头的词元都取出来，然后再将各个词元的检索结果合并，就可以实现 1 个字符的检索了 [3]。

> [3]也有与使用了 uni-gram 的倒排索引相结合的实现方法。

获取以特定字符开头的词元的列表

与按照文档大小对检索结果排序时相同，我们首先还是要做好执行新查询的准备工作。

```
typedef struct _wiser_env {
  ...
  sqlite3_stmt *token_partial_match_st;
  ...
} wiser_env;

init_database(wiser_env *env, const char *db_path)
{
  ...
  sqlite3_prepare(env->db,
    "SELECT token FROM tokens WHERE token like ? || '%';",
    -1, &env->token_partial_match_st, NULL);
  ...
}

fin_database(wiser_env *env)
{
  ...
  sqlite3_finalize(env->token_partial_match_st);
```

```
  ...
}
```

SQL 中的“||”是用于连接字符串的二元操作符。也就是说，我们可以通过“||”把“%”连接到传入的字符串后面。

```
/**
 * 获取与给定的字符串部分匹配的词元的编号的列表
 * @param[in] env 存储着应用程序运行环境的结构体
 */
int
token_partial_match(const wiser_env *env, const char *query,
                    int query_len,
                    UT_array *tokens)
{
  int rc;
  sqlite3_reset(env->token_partial_match_st);
  sqlite3_bind_text(env->token_partial_match_st, 1, query, query_len,
                    SQLITE_TRANSIENT);
  while ((rc = sqlite3_step(env->token_partial_match_st)) ==
         SQLITE_ROW) {
    char *title = (char *)sqlite3_column_text(env->token_partial_match_st,
                  0);
    utarray_push_back(tokens, &title);
  }
  return 0;
}
```

同样不要忘记将该函数的原型声明添加到 database.h 中。

```
int token_partial_match(const wiser_env *env, const char *query,
                        int query_len, UT_array *token_ids);
```

这样就可以获取以某个字符开头的词元列表了。接下来只要再将这些词元所对应的检索结果合并，就可以得到最终的检索结果了。

合并检索到的结果

在 search.c 的函数 search() 中，我们设定了一旦查询字符串的长度小于词元的最大长度，就抛出一条错误信息。

```
if (query32_len < env->token_len) {
  print_error("too short query.");
} else {
```

现在，我们要将这段逻辑修改成下面这样。

```
if (query32_len < env->token_len) {
    char **p;
    UT_array *partial_tokens;

    utarray_new(partial_tokens, &ut_str_icd);
    token_partial_match(env, query, strlen(query), partial_tokens);
    for (p = (char **)utarray_front(partial_tokens); p;
         p = (char **)utarray_next(partial_tokens, p)) {
      inverted_index_hash *query_postings = NULL;
      token_to_postings_list(env, 0, *p, strlen(*p), 0, &query_postings);
      search_docs(env, &results, query_postings);
    }
    utarray_free(partial_tokens);
} else {
```

首先，调用刚刚定义好的函数 token_partial_match()，获取以给定的查询字符串开头的词元的列表。然后，再通过循环将词元从列表中逐一取出，并通过函数 token_to_postings_list() 获取各自的倒排列表。最后，用获取的倒排列表进行文档检索，并通过函数 search_docs() 将检索结果合并到一起。

专栏

如何实现相似文档的检索

在检索的过程中，我们可以把文档看作是词元的集合。同样，也可以把词元相对较少的查询看作是词元的集合。也就是说，我们可以认为，所谓检索就是在这两个词元的集合间进行相似检索。其中的一个词元集合是查询，另一个是作为检索对象的文档。例如，只要将文档作为一个查询发送给搜索引擎，就可以通过倒排索引实现能查找出相似文档的相似文档检索了。

当搜索引擎接收到的查询是一个文档时，如果要检索出包含了查询

中所有词元的文档，就需要大量的处理，而且这样做对于“相似”的界定也未免有些过于严格了。不过，例如通过检索只包含了查询中TF-IDF 值较高的词元的文档，就可以使用较少的处理找出特征相似的文档了。

6-5　调整控制倒排索引更新的缓冲区容量

在启动 wiser 时，我们可以指定控制倒排索引更新的缓冲区容量。缓冲区容量是一个数值，决定了可在内存上临时创建的倒排索引的大小。缓冲区的容量越大，合并倒排索引的次数就越少，构建倒排索引的速度也就越快，但是内存的使用量也会增大。

下面就让我们来确认一下调整缓冲区容量所引起的变化吧。

确认由缓冲区容量的差异带来的不同效果

本节的练习内容就是仅调整参数 t 的操作。例如，我们试着分别执行下面的 3 条命令，并比较它们的执行速度。

示例

```
> time ./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 10 threshold_
（省略了中间的若干行输出）
./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 10 threshold_10.db
204.61s user 24.47s system 99% cpu 3:49.90 total

> time ./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 100 threshold
（省略了中间的若干行输出）
./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 100 threshold_100.db
85.76s user 14.88s system 99% cpu 1:41.08 total

> time ./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 500 threshold
（省略了中间的若干行输出）
./wiser -x zhwiki-latest-pages-articles.xml -m 1000 -t 500 threshold_500.db
70.46s user 9.29s system 99% cpu 1:20.11 total
```

在为 1000 个词条构建倒排索引时，根据缓冲区容量的不同有如下差异。

- 缓冲区最多可存放 **10** 个词条：**204** 秒
- 缓冲区最多可存放 **100** 个词条：**85** 秒

- 缓冲区最多可存放 **500** 个词条：**70** 秒

由此可以看出，较小的缓冲区会花费惊人的时间；而一旦增大了缓冲区的容量，执行时间就会缩短。这是由于在构建倒排索引时，需要先从数据库中获取倒排列表，然后将其与缓冲区上的倒排列表合并，最后将合并后的结果写回数据库。在该过程中，处理次数的不同导致了执行时间上的差异。

用 **sar** 命令分析负载

获取、存储倒排列表的过程会产生大量的 I/O 操作，而在合并倒排列表的时候，CPU 又会忙碌地运转起来。

那么，我们就通过 sar 命令来分析一下 CPU 和 I/O 的负载吧。只要执行下面这条命令，就可以分析出 CPU 和磁盘 I/O 的负载了。

示例

```
> sar -du 1 100 1

17:29:30 %usr %nice %sys %idle
17:29:30   1     0    2   97

17:29:30 device r+w/s blks/s
17:29:30 disk0    83    2184
17:29:30 disk2    0     0
```

%idle 下的数字越小说明 CPU 的负载越高。r+w/s（Mac）或者 rd_sec/s 和 wr_sec/s（Linux）下的数字越大说明磁盘 I/O 的负载越高。

请诸位也亲自用 Wikipedia 的索引检索一番，并计算一下 CPU 和磁盘 I/O 的负载吧。同时我们还能由此验证出启用了压缩后，虽然 CPU 的负载升高了，但是 I/O 的负载却下降了这一事实。

6-6　调整只有英文字母的词元的分割方法

如何避免用英文单词检索时准确率下降的问题

在 wiser 中句子的分割采用的是 bi-gram，但是使用了由 bi-gram 构成的倒排索引的检索，其准确率（将在 7-1 节讲解）通常都不会太高。特别是在用英语检索时，这个问题就会更加明显。作为避免该问题的方法之一，我们可以像下面这样针对不同类型的字符调整词元的分割方法。

- 当遇到英文字符时

  → 将非英文字符出现前的连续出现的所有英文字符作为一个词元

- 当遇到非英文字符时

  → 和原先一样，用 **bi-gram** 分割词元

下面，我们就来尝试实现这个方法。

如何判断某字符是否属于索引对象

\ 在分割词元时，函数 wiser_is_ignored_char() 的作用是判断给定的字符是否属于索引对象。那么，该函数是如何处理的呢？

```
/**
 * 检查传入的字符（UTF32）是否不属于索引对象
 * @param[in] ustr 传入的字符（UTF32）
 * @return 是否是空白字符
 * @retval 0 不是空白字符
 * @retval 1 是空白字符
 */
static int
wiser_is_ignored_char(const UTF32Char ustr)
{
  switch (ustr) {
  case ' ': case '\f': case '\n': case '\r': case '\t': case '\v':
  case '!': case '"': case '#': case '$': case '%': case '&':
  case '\'': case '(': case ')': case '*': case '+': case ',':
  case '-': case '.': case '/':
```

```
  case ':': case ';': case '<': case '=': case '>': case '?': case '@':
  case '[': case '\\': case ']': case '^': case '_': case '`':
  case '{': case '|': case '}': case '~':
  case 0x3000: /* 全角空格 */
  case 0x3001: /* 、 */
  case 0x3002: /* 。 */
  case 0xFF08: /* （ */
  case 0xFF09: /* ） */
  case 0xFF01: /* ！ */
  case 0xFF0C: /* ， */
  case 0xFF1A: /* ： */
  case 0xFF1B: /* ； */
  case 0xFF1F: /* ？ */
    return 1;
  default:
    return 0;
  }
}
```

由于在本书中我们所使用的文本含有 Wikipedia 特有的 Wiki 标记，所以其中会混有各种各样的符号 [4]。

[4]关于中文标点符号的 Unicode 编码可参考 http://www.unicode.org/chars/PDF/U3000.pdf 和 http://www.unicode.org/chars/PDF/UFF00.pdf。——译者注

在 wiser 中，我们放弃了如实地解析 Wiki 标记，而是选择忽略在 Wikipedia 的标记法中使用的各种符号。之所以这样处理，是为了避免大量出现的 Wiki 标记导致构建索引时负载升高。基于同样的原因，我们还会将“，”“。”这样的标点符号排除在索引对象之外。这也算是一种停用词（Stop Words）。有关停用词的详细解释请参考第 7 章。

修改负责分割词元的函数

token.c 中的函数 ngram_next() 的作用是用 bi-gram 分割词元。每次调用该函数，我们都能取出一个含有 *n* 个字符的词元，除非遇到了空格或是指针 p 指向了字符串的末尾。

```
/* 取出含有n个字符的词元，除非遇到了非索引对象的字符或是指针p到达了字符串的结尾 */
for (i = 0, p = ustr; i < n && p < ustr_end && !wiser_is_ignored_char(*p);
p++) {
}
```

下面我们试着对该函数做如下的修改。

```
if (wiser_isalpha(*ustr)) {
    /* 将连续出现的英文字母视作一个词元 */
    for (p = ustr; p < ustr_end && wiser_isalpha(*p); p++) {
    }

    /* 将最后一个英文字母之后的一个字符当作下一个词元的起始字符 */
    *next = p;
} else {
    /* 取出含有n个字符的词元，除非遇到了空格或英文字母，或是指针p到达了字符串的末
    for (i = 0, p = ustr;
       i < n && p < ustr_end && !wiser_is_ignored_char(*p) && !wiser_isalph

    }
    *next = ustr + 1;
}
```

在分割词元时，如果第一个字符是英文字母，就将从这个字母开始一直到最后一个英文字母为止的这一整段英文字母作为 1 个词元处理。否则，就还按原来的方法处理，即取出由 n 个字符构成的词元。

函数 wiser_isalpha() 是个简单直观的函数，其实现如下所示。

```
/**
 * 检查传入的UTF32的字符是否是英文字母
 * @param[in] ustr 输入的字符（UTF-32）
 * @return 是否是英文字母
 * @retval 0 不是英文字母
 * @retval 1 是英文字母
 */
static int
wiser_isalpha(const UTF32Char ustr)
{
  if (('A' <= ustr && ustr <= 'Z') ||
      ('a' <= ustr && ustr <= 'z')) {
    return 1;
  } else {
    return 0;
  }
}
```

可以看出，在这里我们仅仅对英文字母进行了特殊处理。其实对于带有附加符号（Diacritical Mark）（如变音符（Umlaut）等）的拉丁字母也应该进行特殊处理。而且，与英文字母一样，数字和符号等字符也不属于 N-gram，所以在取出时应该将连续出现的数字或符号等算作一个词元。

6-7　确认压缩的效果

观察**Golomb** 编码的效果

在 wiser 中，默认情况下会将经过 Golomb 编码压缩后的倒排列表存储到数据库中，不过在启动时指定了参数“-c none”，即可禁用压缩。

下面，就让我们通过改变参数 -c 的取值，来验证一下 Golomb 编码所带来的压缩效果吧。

示例

```
> ./wiser -c none -x zhwiki-latest-pages-articles.xml -m 1000 wikipedia_100
```

在这里我们依然可以使用 sar 命令来对比压缩启用前后构建索引的负载。构建完成后，在分别用经过压缩和未经过压缩的索引进行检索时，还可以再比较一下磁盘 I/O 和 CPU 的使用率。

对比压缩启用前后的索引大小

前面已经讲过，压缩索引可以减少磁盘的 I/O。下面我们就再来看一下存储压缩后的索引究竟可以节省多少存储空间吧。

索引的大小可以通过查询 SQLite 来获取。因此我们先指定创建好的索引数据库，调出 SQLite 的交互环境。

示例

```
> sqlite3 wikipedia.db
```

倒排列表存储在 tokens 表的 postings 字段中。通过执行以下的 SQL 语句，即可获取到以字节为单位的倒排列表的大小，其中用到了能够返回列中字符串长度的函数 LENGTH() 和对列中数据求和的统计函数 SUM()。

示例

```
> sqlite
> SELECT SUM(LENGTH(postings)) FROM tokens;
```

专栏

避免滥用全文搜索引擎

在调研要采用哪种全文搜索引擎时，一般都要考虑是否还有其他代替方案。

正如本书所介绍的，全文搜索引擎的确可以利用索引提高文档检索的效率。但是索引的大小会随着文档数量的增多而变大，索引大小一旦变大，就需要更大的内存或存储器等存储设备，这就意味着维护检索服务的成本也会随之增大。

近几年，出现了一些只需要借助基于 HTTP 的 API，用户就可以轻松使用的全文搜索引擎。由于在服务中引入这类引擎的过程非常简单，所以笔者也非常能理解那种想要轻轻松松地引入引擎的渴望。但是，如果再重新审视一下服务中的需求，就会发现有时不使用“全文搜索”一样能满足需求。

例如，试着考虑一下提供商品买卖服务的场景。假设需求是“输出某个商家正在出售的商品列表”。那么，在这种场景下，由于每件商品都拥有用文本表示的商家名称，所以此时我们的确可以使用全文搜索引擎来筛选每个商品。

商品 **ID**	商家名称
00001	A 公司
00002	B 公司
00003	A 公司

00004	A 公司
00005	C 公司

例如，用“A 公司”进行检索，就能检索出“A 公司”的商品。

除此以外，我们还可以在 RDBMS 上为每个商家分配一个 ID，然后将商家 ID 关联到商品上，最后再通过这个 ID 进行检索。

商品 **ID**	商家 **ID**
00001	1
00002	2
00003	1
00004	1
00005	3

只要用“1”在商家 ID 中检索，就可以检索出“A 公司”出售的商品了。

这样不但数据量要少得多，而且也不需要使用全文搜索引擎了。也许诸位会认为“一般也不会像第一种方案那样做啊”。但事实好像一旦有了更加方便的系统，人们就会情不自禁地去使用。因此，笔者也曾不止一次地遇到过像第一种方案那样使用了全文搜索引擎的案例。

更进一步讲，也许有人认为第一种方案使用起来更加方便，因为即

使只检索“A”，也能得到有关“A 公司”的检索结果。虽然这种想法并没有错，但是最终结果与为此投入的必要资源是否匹配还是值得研讨的。

全文搜索是一种不需要预先设定用户将会输入的查询即可进行搜索的优秀解决方案，但是在它的实现过程中却需要花费大量的资源。也可以这样夸张地来理解：作为信息检索方法的全文搜索，就是一种不到万不得已时不要使用的非常手段。正所谓“杀鸡焉用牛刀”。但是，对于真正需要的场景，也不必畏缩，请果断地使用全文搜索吧。

第 7 章　为今后更加深入的学习做准备

本书汇总了搜索引擎的基础知识以及提供由搜索引擎支撑的服务时所需要的最基本的知识。尽管如此，显然还有很多技术和知识在本书中并没有涉及。

因此，我们会在本章简单地介绍一下这些技术和知识。如果诸位还想更加深入地学习这部分内容，建议去查阅一些有关搜索引擎的专业书籍。想必读过本书后，诸位一定能顺利地读懂那些专业书籍。

7-1　wiser 没能实现的功能

首先，我们来看一些比较深入的、在 wiser 中没能实现的功能。

倒排索引之外的全文搜索索引

除了本书所涉及的倒排索引以外，在全文搜索中还有各种各样的索引。例如，对字符串的后缀进行排序，使得子字符串可供检索的 Suffix Array。还有作为由 Suffix Array 发展而来的索引结构，近年来倍受瞩目的 FM-index 和 Compressed Suffix Arrays（CSA）等。关于这些技术的详细内容，请参考《字符串高速解析的世界：数据压缩·全文搜索·文本挖掘》（冈野原大辅 著，岩波书店），这本书可称得上是有关最前沿的全文搜索的指南 [1]。

> [1]原书名为《高速文字列解析の世界——データ圧縮·全文検索·テキストマイニング》，目前还没有类似的中文书籍。——译者注

高效处理大规模数据的存储器

在 wiser 中，我们将 SQLite 作为存储器使用，但这未必是最佳的方案。因为在大多数情况下，全文搜索引擎所处理的都是大规模的数据。为了提高处理效率，我们需要设法优化用于存储索引的数据布局以及对于该布局的输入输出方法。因此 Groonga 和 Apache Lucene 等搜索引擎框架中的倒排索引都采用了私有的数据布局来存储索引。

利用缓存提高检索的速度

在进行检索时，搜索引擎要从磁盘等二级存储器中加载倒排列表，不过只需先将经常加载的倒排列表缓存到内存中，有时就可以避免每次检索时都从磁盘加载了。而且，只要先将检索结果本身缓存起来，当再遇到同样的查询时，就可以省略掉检索处理的环节了。

由于我们将对缓存（缓冲区）的管理委托给了操作系统，所以在 wiser 中并没有明确地进行有关缓存的操作。也就是说，我们并没有利用各查询的检索频率和对倒排列表的访问频率等搜索引擎所固有的信息，这意味着在 wiser 中还有大量的地方可以优化。

使用各种各样的压缩方法

在 wiser 中，我们借助 Golomb 编码实现了倒排列表的压缩。除此以外，其实还有在第 5 章讲解过的 variable-byte 编码以及将在附录部分介绍的 Simple 9 和 PForDelta 等各种各样的压缩方法。由于压缩整数序列既是一个非常普遍的问题，又有着广泛的应用范围，所以一直以来研究人员都在提出各种各样的压缩方法。

在选择压缩方法的时候，需要权衡压缩的利弊。例如，在检索处理的过程中，虽然使用压缩率较高的 Golomb 编码相对于不压缩，能够减少输入输出的数据量（以及占用的存储空间），但是解码时却会因此产生额外的 CPU 负载。所以在 CPU 的时钟频率相对低于输入输出带宽的硬件环境中，为了提升速度而使用压缩有时并不是一个好方案。因此，我们需要根据目标和使用的硬件环境来选择合适的压缩方法。

优化搜索结果的排名

在 wiser 中，我们将由文档内容计算得出的 TF-IDF 值作为了检索结果排名的依据。另外，我们还在第 1 章提到了向量空间模型中的余弦相似度和 Okapi BM25。除此之外，还在第 6 章介绍了像 PageRank 那样的由文档间的链接结构计算出的排名指标。

检索结果的排列顺序要“排得好”，很大程度上取决于检索服务以及用户的需求，因此并没有正确的答案。于是，我们在提供检索服务时，就要考虑这些因素并将上述指标结合起来，并以不断优化排名结果为目标。

近几年，研究人员正在开展有关 Learning to rank 的研究工作，这是一种让系统基于来自搜索引擎用户的反馈学习如何排名的方法。

调整准确率和召回率

准确率（Precision）和召回率[2]（Recall）是两个能够定量评价全文搜索结果的指标。

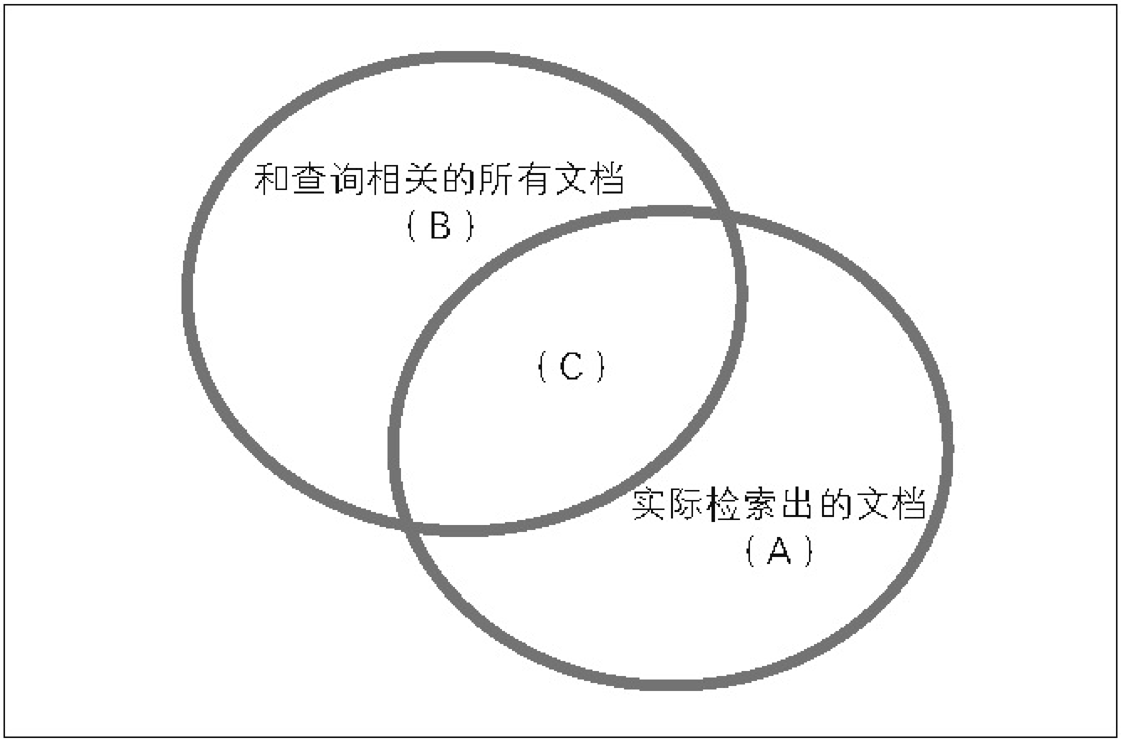
> [2]也称为查全率。——译者注

假设搜索引擎的用户用某个关键词进行了检索后检查了所有的文档，并

将这些文档分成了以下 3 组。

- **A** 组：实际检索出的文档的集合
- **B** 组：和查询相关的文档的集合
- **C** 组：既属于 **A** 组也属于 **B** 组，即和查询相关而实际也确实检索出的文档的集合

准确率和召回率就可以用这 3 个文档的集合来定义。具体的计算方法如图 7-1 所示。



准确率 $= \dfrac{C}{A}$ （在实际检索出的文档中，符合查询的文档所占的比率）

召回率 $= \dfrac{C}{B}$ （在和查询相关的所有文档中，检索出的文档所占的比率）

图 **7-1**　准确率与召回率

从这两个数值可以很直观地看出，如果准确率较低，则说明“出现的都是些和查询无关的检索结果”。反过来，如果召回率较低，则说明“对于某个查询，明明应该检索出某某结果的，实际上却几乎看不到这样的结果”。无论是哪一种情况，都是我们不愿意看到的。

准确率与召回率一般是无法兼得的。也就是说，如果其中的一个值提高了，另一个值就会降低。例如，我们来考虑这样一种极端情况。假设所有文档都作为检索结果返回了，那么此时虽然准确率极低，但是召回率

却高达 100%。可这样的搜索引擎会有实用价值吗？由于根本就没有进行任何筛选过滤，所以能否称其为搜索引擎都是值得怀疑的。

要想提升准确率，就需要从检索结果中剔除不相关的文档。但是，由于判断“是否相关”本来就很困难，所以在这个过程中，免不了会将那些本应该是相关的检索结果也剔除掉。因此，我们需要根据搜索服务的性质及用途对准确率和召回率做出适当的调整。

改变词元的提取方式是调整准确率和召回率的方法之一。虽然在本书中我们是使用 N-gram 将句子分割成词元的，但是只要改用词素解析的方式，一般就可以提升准确率。

例如，假设用户想要了解“华山”的信息。那么，如果只从字面上来看，在检索“华山”时，检索出了含有“九华山”的文档也是有可能的。但是，从意思上来看，“九华山”和“华山”又是完全不同的。由于词素解析会将句子分割成有意义的单位（词素），所以即便是检索“华山”，也检索不出含有“九华山”的文档。

也就是说，词素解析与 N-gram 一般会呈现出如下的关系。

- 相对于 **N-gram**，使用词素解析能够提升准确率

- 相对于词素解析，使用 **N-gram** 能够提升召回率

请诸位务必挑战一下通过词素解析分割词元的过程，感受一下准确率和召回率间的关系。

降低检索结果排序处理的负载

面对大量的检索结果，对其进行排序本身就会产生巨大的负载。由于一般的搜索引擎只会呈现最前面的 K（K = 10~100）条检索结果，所以只需要对前 K 条检索结果排序即可。也就是说没有必要像现在的 wiser 那样对全部检索结果都进行排序。

这样的话，就会经常用到堆这种数据结构，以提升解决 Top-K Sort 问题，即对前 K 条检索结果排序的效率。例如，我们可以通过 C++ 中的 std::partial_sort，Perl 中的 Sort::Key::Top 以及 SQL 中的方言 SELECT TOP(K) 或 ORDER BY col LIMIT K 等来提升解决 Top-K Sort 的效率。

另外，请诸位注意这样一点，如果想从第 91 条结果开始呈现出 10 条结果的话，就要先对前 100 条结果排序，然后再从第 91 条结果开始取出 10 条结果。也就是说，此时 K 的取值是 100 而不是 10。

并行处理

在 wiser 中，无论是构建倒排索引，还是使用倒排索引进行检索，都是在 1 台计算机上进行的。由于 1 台计算机能处理的索引规模是有限的，所以要想使用大规模的索引，就需要使用多台计算机对那些索引进行并行处理。关于并行处理的细节请参考附录。

结合对属性的筛选过滤

有些搜索服务会结合全文搜索和属性值匹配两个维度对文档进行筛选过滤。例如，在商品检索中，就经常可以看到能够指定价格范围以配合关键词检索的实例。

在上述场景中，全文搜索引擎需要具备根据属性进行筛选过滤的功能。而且，在对文档进行筛选过滤的过程中，全文搜索引擎还需要正确地判断出是先进行全文搜索好，还是先进行属性值匹配好。例如，相对于全文搜索，通过属性值匹配能够更好地筛选过滤文档时，就应该先按照属性值检索文档，再对相应的文档集合进行全扫描等操作，以此加快全文搜索的速度。

分面搜索

诸位在购物网站上进行检索时，有没有过因为查找到的商品过多而感到厌烦的情况呢？所谓分面搜索（Faceted Search）就是一种针对检索结果，检索每一个属性值，然后再将各个属性值对应的结果数呈现出来的技术。例如在购物网站中检索“服装”，就可以看到如下所示的用于筛选过滤的属性值，以及用各个属性值检索出的结果数。

- 图书（**1360**）
- 服饰箱包（**397**）
- 母婴用品（**92**）

Groonga 和 Apache Lucene 等都提供了能够轻松进行分面搜索的功能和相应的 API。

专栏

时延和吞吐量

面对来自大量用户的各种各样的查询，我们希望 Web 上的检索系统能够快速地呈现出检索结果。

这里所说的“快速”有两层含义：一个是时延小；一个是吞吐量高。

时延指的是从接收到处理请求到将处理结果返回给请求者的时间。大多数人一听到“快速”可能就会联想到时延小。另一方面，吞吐量指的是在一定时间内能够处理的请求量。

一般来说，时延和吞吐量是无法兼得的。也就是说，将其中一方调优了，另一方就会恶化。例如，试想在单核 CPU 的计算机上进行检索处理。检索处理分为 CPU 处理和 I/O 处理两部分。在一般情况下，进行 I/O 处理时，CPU 是处于空闲状态的。因此，在查询 1 进行 I/O 处理时，如果收到了一个来自查询 2 的请求，那么此时 CPU 可以先进行查询 2 的 CPU 处理。像这样，通过用 1 个 CPU 并行地处理多个请求，就应该可以提升吞吐量。

接下来，假设在进行查询 2 的 CPU 处理时，查询 1 的 I/O 处理结束了。那么，由于此时 CPU 还在执行对查询 2 的处理，所以从此刻到查询 / 恢复 CPU 处理之时会产生一段等待时间。也就是说，在本例中，我们是通过牺牲查询 1 的时延来换取系统吞吐量的提升的。

7-2　全文搜索引擎 Groonga 的特点

全文搜索引擎 Groonga 是一款笔者（末永）也参与了开发的开源全文搜索引擎。本节我们就来了解一下 Groonga 的特点吧。

通过词元的部分一致检索提升召回率

在上一小节讲解准确率和召回率时，我们说过，用户并不希望在检索“华山”时看到包含“九华山”的文档。但是，万一用“华山”连一个文档都检索不到时，提供给用户包含“九华山”的检索结果也不失为一种对策。

在 Groonga 中，当检索结果小于一定数量时，Groonga 就会进行两种附加检索。第一种附加检索是把查询字符串本身当作一个词元进行的检索。以“北京大学”这个查询字符串为例，在大多数情况下，Groonga 还是会通过词素解析等手段将其分割为“北京”和“大学”两个词元，然后再分别对二者进行检索。而一旦检索结果的数量不足，Groonga 就会把“北京大学”当作是 1 个词元进行附加检索。

根据上下文的不同，词素解析器对词元的分割位置也会发生变化。同样是“北京大学”，有时会将其分割为“北京”和“大学”两个词元，有时又会将其解释成是一个词元。不过，通过上述策略即可防止这种由词素解析的不稳定而引起的检索遗漏。

即使进行了上述附加检索，检索结果也依然少于预期的数量时，作为另一种附加检索，Groonga 会进行词元的部分一致检索。在 Groonga 中，开发者们通过半无限长字串（后缀）和前方一致检索的结合实现了词元的部分一致检索。

所谓半无限长字串是指从某个字符串中去掉 0 个以上的起始字符后所剩的字符串。在 Groonga 中，pat.c 中的 sis（即 Semi-InfiniteString 的缩写）实现了有关半无限长字串的处理。“北京大学”的半无限长字串如下所示。

北京大学

京大学

大学

学

假设我们在检索“京大”这个字符串时没有找到任何检索结果。这时 Groonga 会发现“京大学”与“京大”的前半部分是一致的，进而判断出“京大学”是由“北京大学”这个词元产生的半无限长字串，于是接下来就会开始用“北京大学”进行检索。通过这样的对策，即使接收到了比词元还要短的查询字符串，Groonga 也能设法提升召回率。

另外，为了存储词元和半无限长字串，开发者们还在 Groonga 中采用了支持前方一直检索的称作基数字典树（Patricia Trie）的数据结构。

使用内存映射文件

在 Groonga 中，开发者们通过内存映射文 件（Memory-Mapped File）将文件的内容映射到了内存空间（虚拟存储空间）。具体来说就是在 lib/io.c 中，通过调用系统内核函数 mmap() 来使用内存映射文件。

使用内存映射文件既有好处又有坏处。好处是由于可以省略掉从操作系统中的内核空间向用户空间复制不必要的数据的环节，所以在绝大多数情况下都可以加快 I/O 处理的速度。而且，从易于实现的角度来看，对于那些利用了多进程或多线程同时对索引等进行引用的搜索引擎而言，映射了文件的内存空间可以在多个进程间共享这一点还会大大降低实现的难度。

而使用内存映射文件的缺点则在于，当需要将已写入内存的内容同步到文件时，除非明确地调用 msync() 等函数，否则同步工作就会由操作系统来接管。因此，万一在写入文件的过程中发生了进程死掉等异常情况，我们将完全不清楚哪些数据已经写到文件中了。换言之就是，由于很容易发生数据被破坏等异常情况，所以在实现时需要倍加注意。

另外，由于有些操作系统无法提供稳定的内存映射文件功能，所以要在内存和文件间反复进行数据传输。

片段

在很多 Web 检索服务中，都会在网页的标题下方显示一段网页内容的摘要。在检索中，我们将这样的摘要信息称为片段（Snippet）。

在 Groonga 中实现了能够快速生成片段的功能。片段的基本原理就是先从文档中找出检索关键词，然后取出其前后的句子。接下来，还要对包含在句中的检索关键词部分进行高亮处理。

作为进一步的处理，还要对简体字和繁体字，全角字符和半角字符的差异进行归一化处理。这样就能生成片段了。另外，Groonga 还对网页中的 HTML 元字符进行了转义处理（Escape），并且允许开发者为与检索关键词对应的部分赋予任意的 HTML 标签。

有关片段功能的源代码都写在了 lib/snip.c 中，有兴趣的读者不妨去读一读。

专栏

宣传活动的重要性

Groonga 是一款基于 LGPL 的开源软件。由于能够吸引大量用户使用，所以开源软件具有能够根据各行各业用户的反馈迅速进行优化的优点。但是，单单依靠将软件公开给大家使用是无法得到大量反馈的。

因此，为了得到大量的反馈，笔者经常会开展所谓的宣传活动，并奔波于各种开发者云集的技术大会和研讨会，努力向更多的人介绍 Groonga 的代码库。不仅如此，笔者还会走访代码库的使用者，询问他们对哪些地方不满意，并根据这些反馈进行改进。

这些实实在在的努力并没有白费，Groonga 正在逐渐成为被开发者们广泛使用的软件。

7-3　实现出考虑到用户意图的搜索引擎

在运维使用了搜索引擎的服务时，往往会遇到始料未及的麻烦。有时还会接到诸如“检索速度太慢了”“检索不出结果”“太难用了”等来自用户的负面反馈。搜索引擎不能仅仅是速度快，还需要考虑到用户的意图。因此，我们的目标是开发出既能检索又好用的搜索引擎。本节中我们就基于上述观点，向诸位介绍一下笔者（末永）在运维实际的服务时所付出的努力吧。

引入停用词

停用词是指不属于处理对象的词元，通常由在检索对象中频繁出现的单词构成。

例如，如果用“的”“是”这样的单词对中文网页进行检索，那么大多数网页都会成为检索结果。所谓检索，就是为了从大量信息中提取出自己感兴趣的信息而进行的操作，因此可以认为那些会产生大量检索结果的单词都是些“提取信息能力很差”的单词。对于这些单词而言，其倒排列表自然会很长，这就导致了在存储时要花费大量的存储空间，而在扫描时又要消耗大量的 CPU 资源。这些都是我们不愿意看到的，因此需要引入停用词的机制来避免上述问题。

应对词素解析的错误

有些词素解析器会在事先学习大量文档后，才开始进行词素解析。对于一篇文章，先由人对其进行词素解析，然后再由机器拼命地学习解析结果。在这个过程中，通常会用报纸中的新闻等语法生硬的文章作为正确的数据。由于报纸上的文章都是用标准的中文书写的，而且报纸上也不太可能刊登一些语法奇怪的文章，所以只学习了这类文章的严肃的词素解析器会面临什么样的下场呢？由于这种中规中矩的解析器并不能适应像是在博客或微博上常见的那类语法随便的文章，自然也就无法顺利地解析这种文章了。

那么，接下来就让我们看一下应该如何应对词素解析的错误。方法之一就是让词素解析器重新学习由这类语法不规范的文章构成的文档，以成为适应这类文档的词素解析器。也就是说，要将社会上各种风格的中文

文章，也包括那些虽然语法不正确，但却会经常使用的中文，都教给中规中矩的解析器。这就好像是让京剧大师唱流行歌曲一样。当然，在学习的过程中需要大量语法不规范的文档数据，而这些数据都要由人来提供。

另外，还有的词素解析器可以返回多个带有得分的候选解析结果。与使用只返回 1 条解析结果的词素解析器相比，使用这样的词素解析器时，只要对所有能划分出单词边界的得分较高的模式都提取一遍词元，并在构建倒排索引时将这些词元全都用上的话，就可以提升检索的正确率。

> 专栏
>
> 断句错误
>
> 以“乒乓球拍卖完了”这句话为例，可以有如下两种解读方法。
>
> 乒乓球拍 / 卖 / 完了
> 乒乓球 / 拍卖 / 完了
>
> 像这种可以有多种解读方法的句子就是“会出现断句错误的句子”。这样的句子从根本上就难以通过词素解析正确地分割出词元。对于这类句子，就连人们都不能作出统一的解释，更何况是机器了，不能进行正确地分割也就是理所当然的。

处理全角字符和半角字符

如果用全角字符和半角字符检索出的结果不一样，那么大多数用户都会觉得这样的系统很难用。因此，我们需要使全角字符和半角字符的检索结果保持一致。

“虽然表示字符的编码不同，但是字符本身却是相同的”，为了达到这个效果，我们需要对字符进行某种归一化处理。Unicode 字符编码标准定义了如下 4 种被称为“Unicode 归一化”的文本归一化处理过程。

- **NFD**
- **NFC**
- **NFKD**

- **NFKC**

其中 NF 是 Normalization Form 的缩写，即归一化形式；D 是 Decomposition 的缩写，即分解；C 是 Composition 的缩写，即合成；K 代表 Compatibility，即互换性。

“K 不是缩写啊”，也许会有人产生这样的疑问，其实这是由于如果还用 C 表示 Compatibility 的话就会和代表 Composition 的 C 发生冲突，所以这里用 K 代替了 C。

这些都是特性不同的归一化形式，在全文搜索上使用 NFKC 即可达到较好的效果。在 NFKC 中，我们首先要将构成字符串的字符从字符串中一个个地分解出来。然后，在构成字符串的字符中，我们还要对那些有如全角、半角以及简体、繁体等多种表现形式的字符归一化为特定的形式。如果是英文字符就归一化为半角字符，如果是汉字就归一化为简体字。最后我们还要将分解后的一个个字符组合到一起。

在各种编程语言中，有些已经以标准组件的形式提供了 NFKC 的代码库。而且，IBM 还引领开发了用于处理 Unicode 的代码库 ICU[3]。如果诸位使用的编程语言中没有集成这样的组件，那么可以借助语言绑定技术（Language Binding）实现从各种编程语言中调用 ICU。

> [3]ICU 的官方网站为 http://site.icu-project.org/。

对查询进行归一化

由于查询是由人输入的，所以在输入的内容中免不了会有偏差。因此，我们就需要像构建索引时对字符串进行归一化那样，对所有的查询也进行归一化。

留意布尔检索的解析过程

在 Web 检索中，用户有时会输入很复杂的检索表达式。例如，有时要排除含有特定关键词的文档，有时是已知多个关键词，需要查找至少包含其中任意一个的文档。这时，很多搜索引擎都支持用类似“- 检索词”的语法指定前一种条件，用类似“检索词 1 OR 检索词 2”的语法指定后一种条件。不仅如此，为了能改变这些检索条件的应用顺序或是对其分组，有的搜索引擎还支持在查询中使用括号，如“搜索引擎 -(Google

OR Yahoo)”。其实，只要认真一些，这些功能实现起来并没有太大的难度。无非是先对查询进行语法解析，然后再按顺序调用相应的处理过程。

但是，大多数搜索引擎都不允许查询中只包含要排除的检索词。例如，可以试试在 Google 上检索“-Google”。该查询的含义是，从所有的文档中检索出不包含 Google 这个词的文档。从含义本身来看并没有什么不合理的地方，可实际结果却是找不到任何检索结果。这是因为这样的查询不仅会因检索结果过多导致实用性下降，还会引发过高的负载。如果要自己制作搜索引擎，一般也可以参考 Google 的做法，即对于只含有要排除掉的检索词的查询，不返回任何结果即可。

对于使用了 OR 的查询，也有值得注意的地方。通常在使用 AND 检索时，随着检索词的增多，检索出的结果会逐渐变少。但是与此相反，使用 OR 检索却能使检索结果越来越多。而且，用户在进行 OR 检索时都拥有很强烈的“想无一遗漏地检索”的愿望，所以他们用 OR 连接起来的词语数量也会变得越来越多。另外，在使用 Web 检索系统对特定公司名或商品进行定点观测时，用户还会定期地执行由 OR 连接起来的多个词语的查询。因此，对于 OR 检索，我们应该通过限制能够连接的单词数来避免出现过多的检索结果，从而减轻检索处理的负载。

通过词素解析器适当地解析查询

通过词素解析将作为检索对象的文档分割成词元时，同样也需要通过词素解析将查询分割成词元。

为了正确地解析出句子中的所有词素，大多数的词素解析器都是经过调整的。但是，用户很少以句子的形式给出查询，多数情况都是只给出了句中的只言片语。面对这样的查询，大多数词素解析器解析起来都会很棘手。

对错误的输入进行修正

由于查询是由人来输入的，所以出现一些细小的错误也是在所难免的。例如，将拼音转换成汉字时选错了汉字，导致输入了同音异义的词语，或者忘记输入了一些字符，等等。如果即便输入了错误的查询，也能正确检索的话，用户就会感到很高兴。

实现输入错误修正的方法之一是事先准备好词典。词典中预先记录了一些可预见的常见错误所对应的正确写法。以数学家的名字为例，如果用户输入的是“傅立叶”就将其修正为“傅里叶”，如果用户输入的是“马尔科夫”就将其修正为“马尔可夫”等。

虽然这样的词典也可以由人工来完成，但是持续维护这样庞大的词典却是件非常麻烦的事。其实编纂词典这种工作也可以用机械的方式来解决。例如，我们只要解析检索出的结果，就可以从中提取出用户连续输入的关键词，以此为依据，应该就可以提取出典型的错误示例了。

作为实现上述方案的方法之一，我们可以利用日志。日志中包含了用户的识别信息、其使用的查询以及检索时间。例如，在比较两个查询时，如果它们的检索时间比某个预定的时间还要短，并且又很相似，那么就可以推测这是用户自己修正了错误的查询。在由此产生的众多候选查询中，通过将大量用户都进行过同样修正的词语收录进词典，就可以导出一份能够自动进行错误修正的候选查询了。

输入补全

很多现代的搜索引擎都提供了查询补全的功能。例如，我们只要输入查询中开头的几个字符，搜索引擎就会以列表的形式提示我们候选的查询。

要想实现这个功能，只需要为经常检索的字符串创建能够进行前方一致检索的结构即可。以“倒排索引”为例，我们就需要提前创建某种结构，使搜索引擎能够根据“倒”“倒排”“倒排索”这样的几个起始字符检索出“倒排索引”。

用于存储字符串集合的数据结构“字典树”，是一种能够快速进行前方一致检索的结构。例如，在存储由“c”“c++”“c#”“java”“javascript”“perl”“php”“python”“r”“ruby”构成的字符串集合时，字典树会使用如图 7-2 所示的结构保存信息。

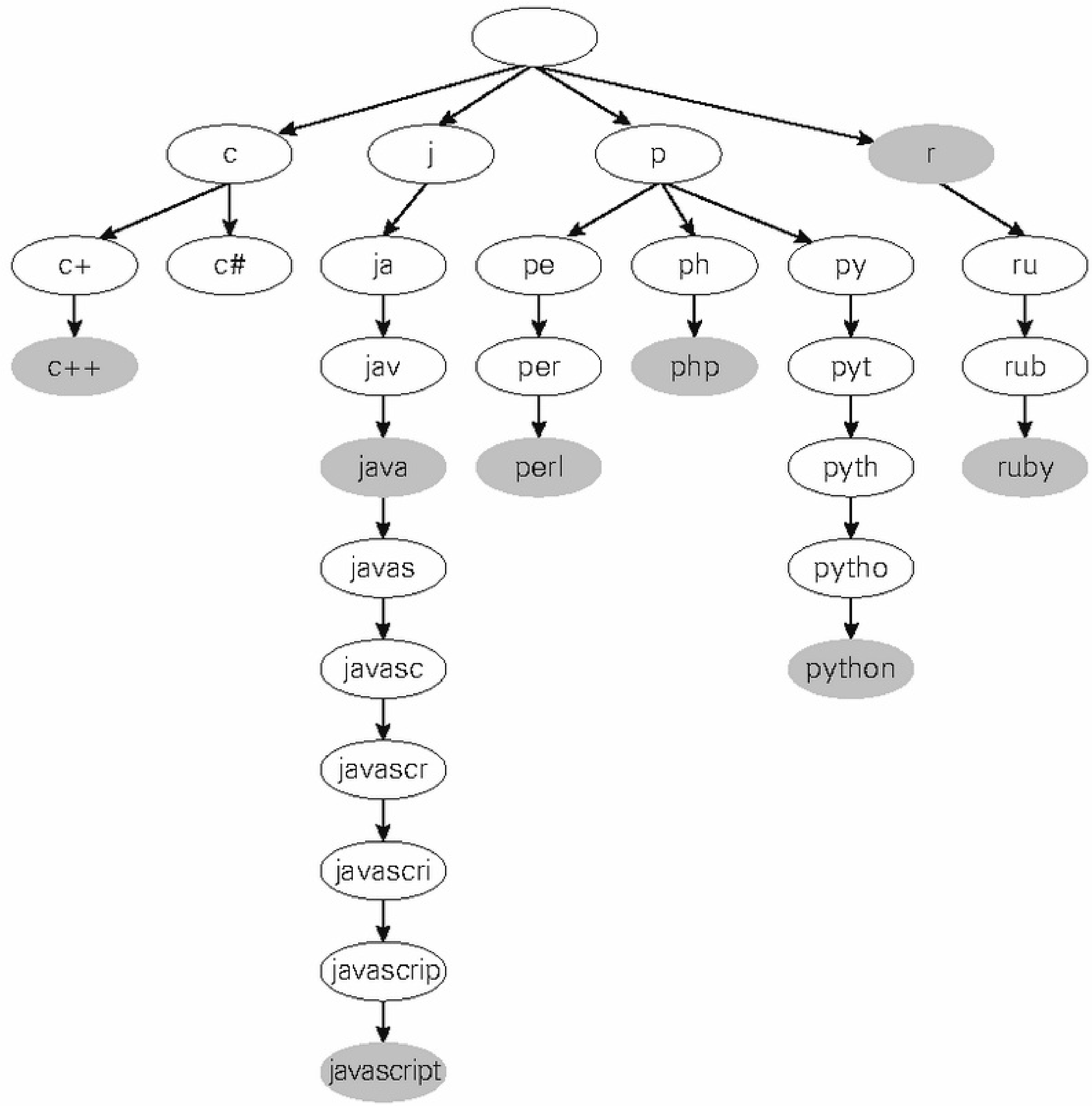


图 **7-2**　由编程语言的名字构成的单词查找树

从图 7-2 可以看出，如果几个字符串以相同的字符开头，那么单词查找树就会用公共的结点来表示这个字符。我们只要用这个结构来保存数据，就可以轻松地获取前方一致的字符串的列表了。

在实际应用中，设计更加巧妙的基数树（Patricia Tree）和后缀数组（Suffix Array）等也是常用的数据结构。作为这些数据结构的实现，只要使用 Darts[4]、Darts-clone[5]、Tx[6] 等优秀的代码库，就可以快速地进行

前方一致检索了。

> [4]工藤拓提供的代码库 darts。http://chasen.org/~taku/software/darts/

> [5]矢田晋提供的、具有和 Darts 相似接口的代码库。http://code.google.com/p/darts-clone/

> [6]冈野原大辅提供的代码库。http://code.google.com/p/tx-trie/

另外，在将补全了的候选字符串提示给用户时，最好先按照某种有意义的标准对其排序。例如，可以按照检索频率对候选结果排序，这样就能让用户更快地选择到最近被多次检索过的查询了。

建议用户检索相关的关键词

诸位是否都遇到过检索结果过多的情况呢？遇到这种情况时，用户经常会为接下来该用哪个查询来筛选检索结果而感到困惑。

于是，在 Google 中就有一个显示“相关的检索关键词”的功能。只要单击列出的关键词，就可以得到经过了更加严格筛选的检索结果。

我们可以在解析用户输入的查询时对其进行统计，然后使用由经常同时检索的多个查询构成的集合来实现该功能。这样的话，在搜索引擎接收到查询后，就可以提示用户有哪些查询会经常和这个查询一起检索了。

7-4　收集、提取文档时的要点

制作爬虫时的处理要点

虽然我们并没有在 wiser 中实现爬虫的功能，但要想实现 Web 检索系统，爬虫的制作必不可少。下面我们就来看一下在制作爬虫时应该注意哪些细节。

❚ 应该如何调整爬虫的访问间隔时间

爬虫会机械地在网络上爬取数据，从而给 Web 服务器带来巨大的访问量。这种行为对于 Web 服务器的管理员来说是很头痛的。因为当一部分服务器被大量访问时，Web 服务器的大部分计算资源都在应对爬虫的访问。

对于爬虫而言，在很短的时间间隔内频繁地访问一部分 Web 服务器也没有什么好处，因为网页的更新往往没有那么频繁，一次又一次地获取尚未更新的网页也只是在做无用功。

为了解决这个问题，很多爬虫都会先将上一次访问网页的时间记录下来，并具备一个先判断再爬取的机制。只有从上一次访问该网页的时间算起，已经过了足够长的时间时，才会再次爬取同一个网页。这里所说的“足够长的时间”到底是多长，要根据网站的更新频率等来推测，并且要对各个网站分别进行设定。只要具备了这种机制，爬虫就不会再对同一个网页频繁地爬取了。

另一方面，如果将爬取的间隔时间设得过长，就有可能无法获取最新的信息了。因此，需要针对重要的和经常更新的网页做一些诸如缩短间隔时间的调整。

另外也有一些高级的爬虫，可以根据网页的更新周期动态地调整爬取的时间间隔。

❚ 反垃圾策略

有一些网页中会包含重复内容。例如，由新闻网站发布的报道除了在新

闻网站本身刊登，还会被其他合作机构的网站转载。之所以这样做，是因为新闻网站可以从转载的网站那里获益，而转载的网站则可以通过刊登新闻来吸引用户，从而实现了双赢。然而，这样做会引发一个“在不同的 URL 上刊登了同一条新闻”的问题。不仅如此，出于各种各样的原因（比如垃圾站点），还存在着很多这种内容重复的网页。

可是，避免爬取这些网页又是很困难的。因为对于完全相同的文章，必须要判断出其中哪一篇才是最先发表的。而又由于网页的发表没有登记制度，所以就需要收集各种证据来判断，例如“有没有指向引用源的链接”等。

作为判断垃圾站点的方法之一，我们可以考虑基于网站的“可疑程度”来判断。例如，由于垃圾站点的目的是放置指向外部网页的链接以及让用户点击广告，所以垃圾站点要么是链接数远远高于文档数，要么是含有过多的广告。我们可以将这样的网页作为“可疑”的网页，并对其采取降低爬虫爬取的优先级等对策，这也不失为是一种好的方法。

在实际的服务中，有时也还会采用不同的处理方法。例如，对于检索结果，通过获取用户实际上有没有点击来计算每个网页的点击率，然后对于点击率较低的网站降低爬虫爬取的优先级，这样就可以利用人们的判断能力使爬虫爬取的优先级达到最优状态了。

进行过上述判断后，我们还可以采取更进一步的优化措施，例如不再爬取在可疑网页较多的域名下发布的网页。

我们最终需要的是将上述几种处理方法结合起来，以创造出一种误检率低且能避免爬取垃圾站点的爬虫。由于垃圾站点每天都在发展进化，所以应对的策略也必须不断地发展进化。不过，凡事都要有个限度，过于严格的对策反而会将本来对用户有用的网站也排除在爬取对象之外。

❙ 对付恶意的 SEO

SEO（Search Engine Optimization）的意思是“搜索引擎优化”，是一种通过创建有针对性的网页，使其在搜索引擎的检索结果中排名上升的行为。

对于爬虫而言，SEO 本身既有好的影响，也有不利的影响。由于 SEO 就是“针对搜索引擎完善内容”，所以这一点可以为爬虫带来一些提示。

例如，可以修正 HTML 语法上的错误，像 <a rel="nofollow"> 这样，将 a 标签的 rel 属性的值设为 nofollow，表示爬虫可以不必顺着该链接爬取。

另一方面，在 SEO 中也有对爬虫不利的影响。一个颇具代表性的例子是创建者在网页中生成了大量的链接。当爬虫遇到含有大量链接的网页时，爬取所有链接所指向的目标网页可能会大幅度地加剧爬虫的负载。从保护爬虫的观点来看，处理这种含有大量链接的 SEO 还是比较简单的。只需要限制从 1 个网页开始爬虫所能爬取的链接数即可。早先进行了这类 SEO 的网页会受到 Web 检索系统的惩罚，或是在检索结果中的排名被降低，或是将其从检索结果中删除，但是最近这样的惩罚却很少见了。除此以外，还有其他的处理方法。例如，暂且将那些存储着含有大量链接的网页的 Web 服务器作为垃圾主机记录在黑名单里，以后就可以不再对其进行爬取了。

❚ 处理通过客户端变更内容的网页

早先的网页都是静态网页。也就是说，那时的浏览器仅仅是一种按照获取的 HTML 文件进行渲染的程序。但是，如今却存在大量能够通过客户端的处理，使用 JavaScript 和 Flash 等技术动态变更内容的网页。

要想直接正确地爬取这些网页，还是比较困难的。虽然也有些爬虫会解析 Flash 的内容，或通过执行简单的 JavaScript 来抽取用于检索的文本。但是为了执行这样的处理，需要一种和实际的浏览器具备相同功能的程序，而这就需要相当大的投入了。就算存在能进行这类处理的程序，执行起来也需要大量的计算资源和时间。而且，这样的程序还必须要能够处理不怀好意的网页，如那些在 Flash 和 JavaScript 中执行了死循环等的网页。

但是，最近也有很多动态网页为了增加来自搜索引擎的流量，采取了一些有利于爬虫的举措。例如，通过 RSS 来 [7] 发布网页中的信息。这样即便爬虫不去直接解析动态网页，也一样能获取到有关网页的信息。

> [7] RSS（RDF Site Summary、Rich Site Summary、Really Simple Syndication 等的缩写）是一种用于发布网站更新信息的文档格式。虽然版本不同会导致该缩写代表的单词不同，但是其本质都是一种用于以计算机易于处理的形式提供网站更新信息的机制。

❚ 估算爬虫所需的必要的资源和时间

对诸位来说，要爬取世界上的所有网页并不现实，因此自然要缩小爬取对象的范围。此时，非常重要的一件事是预估作为爬取对象的网页有多少，以及爬取这些网页需要花费多长时间。

Google 在 2008 年的报告中指出，全世界存在着超过 1 万亿的网页。假设获取 1 个网页需要花费 200ms，那么收集 1 万亿个网页的数据，就需要花费大约 6341 年。即使使用 100 台计算机，也需要花费 63 年之久。另外，假设一个网页的大小是 64KB，那么要存储 1 万亿个网页的数据，就需要约 60PB 的存储空间。都不需要实际地去操作，也能知道这是不可能实现的。

在启动爬虫前，请诸位也尝试着估计一下需要多少资源和时间。这对于设计那些会利用到爬虫爬取数据的服务来说是至关重要的。

在提取文本时需要处理的要点

在提取文本时，也有几个需要处理的要点。

▍对应各种各样的文件格式

除了 HTML 和 XML 等格式以外，世界上还有其他各种各样的文件格式。例如，在诸位所就职的公司的文件服务器上，就应该会有 Word、Excel、PowerPoint 等堆积如山的 Microsoft Office 文件。在大多数情况下，由于这些文件都是用私有的文件格式记录的，所以为了使其能够检索，就需要先理解这些文件格式的细节，然后再提取内容。

▍正确地判断字符编码

由于 Web 上的文件是用各种各样的编码编写的，所以要想提取文件的内容，还需要正确地判断其编码。在中文中，常见的字符编码有 GB2312、GBK 和 UTF-8 等。只要有了一定量的文档，就可以推测出某个文档使用的是哪种字符编码了。

在诸位所使用的编程语言中，也有如下所示的能够自动判断字符编码的函数或模块。

- **PHP：mb_detect_encoding** 函数

- **Ruby：rchardetl9** 模块
- **Java：juniversalchardet** 项目
- **Python：chardet** 模块
- **C** 语言：**universalchardet**

比起自己应对所有处理，使用这些函数或模块，可以以更高的精度来判断字符编码。

另外，对于网页（Web 上的文件）来说，有时 HTTP 响应头中的 Content-Type 字段会标识出该页面（文件）的字符编码。我们也可以将其作为提示来判断页面（文件）的字符编码。

附录

A-1　深度话题

近几年的压缩方法

近几年，研究人员不断提出了若干种新的编码方法。新的编码方法不但能达到较高的压缩率，还能实现快速解码。在本节，我们将介绍其中的两种编码方法——Simple9 和 PForDelta。

▌Simple9

Simple9（以下简称为 S9）是一种通过将尽可能多的整数存储到 32 比特中来实现压缩的编码方法。S9 会将 32 比特的空间分割成前 4 比特（高 4 位）和后 28 比特（低 28 位）两部分，前 4 比特用于存储表示整数存储模式的状态编号，后 28 比特用于存储基于此存储模式的整数。

在整数存储模式中，预先定义了如下 9 种模式。

- **28** 比特 **× 1** 个
- **14** 比特 **× 2** 个
- **9** 比特 **× 3** 个
- **7** 比特 **× 4** 个
- **5** 比特 **× 5** 个
- **4** 比特 **× 7** 个
- **3** 比特 **× 9** 个
- **2** 比特 **× 14** 个
- **1** 比特 **× 28** 个

例如， 对于像 [1，3，8，2，16，9，5] 这样的 7 个整数都不大于 16 的整数序列，通过应用“4 比特 ×7 个”的存储模式，即可用 32 比特来表示。

而在解码时，则要先看一下待解码数据的开头部分，以判断其所应用的存储模式，从而应用针对各存储模式预先硬编码好的解码函数进行解码。通过预先将位操作进行硬编码，即可在对定长比特进行解码时避免分支操作，从而提升解码操作的速度（由于程序中没有分支操作，所以处理器的流水线处理（Pipleline）并不会发生阻塞（Stall））。

PForDelta

与 S9 一样，PForDelta[1] 也是一种通过将整数序列压入定长比特数组来实现压缩的编码方法。但是，PForDelta 在压入大量整数时的处理与 S9 有所不同。

> [1]编码方法 PForDelta 最早由 Zukowski 等人（参考文献 5）提出，之后又由 Yan 等人加以优化。本书所讲解的是经过优化的 PForDelta。

PForDelta 首先会将整数序列分割成若干个区块，每个区块中可存放 32 的倍数（这里假设是 128）个整数。接着要对各个区块分别求出一个能够容纳下该区块中 90% 的整数的比特宽度 b。

接下来，我们要将这 90% 的整数填入“b 比特 ×128 个”的数组（以下称为数组 A）中。当然，由于用 b 个比特无法存储剩余 10% 的整数，所以还要将这些整数当作特殊数值来处理。对于要特殊处理的数值，我们只将其最后的 b 个比特存储到数组 A 中，而将其前面的比特存储到另一个数组（数组 B）中。另外，还要再用另外一个数组（数组 C）来存储哪个整数需要特殊处理。此时，我们可以使用 S9 等编码方法对数组 B 和数组 C 进行压缩。

用 PForDelta 进行编码的示例如图 A-1 所示。

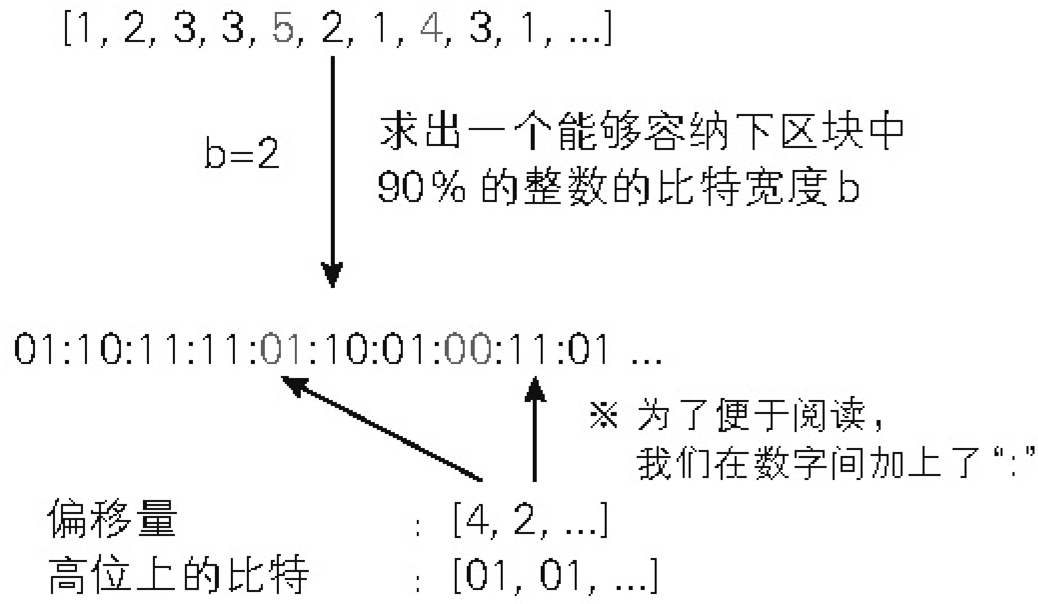


图 **A-1**　用 **PForDelta** 进行编码的示例

PForDelta 的特点是解码速度极快。Zukowski 在论文中介绍的解码方法如下所示。

代码清单　**PForDelta** 的解码方法

```
int Decompress<ANY>( int n、int b,
    ANY *__restrict__ output,
    void *__restrict__ input,
    ANY *__restrict__ exception,
    int *next_exception )
{
    int next、code[n]、cur = *next_exception;
    UNPACK[b](code、input、n); /* bit-unpack the values */
    /* LOOP1: decode regardless */
    for(int i=0; i<n; i++) {
        output[i] = DECODE(code[i]);
    }
    /* LOOP2: patch it up */
    for(int i=1; cur < n; i++、cur = next) {
        next = cur + output[cur] + 1;
        output[cur] = exception[-i];
    }
    *next_exception = cur - n;
    return i;
}
```

解码由 2 个循环分为 2 个阶段完成。在第 1 个循环中，要从 b 比特的定长数组中将各个整数解码。而在第 2 个循环中，要将高位上的比特填补到需要特殊处理的整数上。

由于第 1 个循环中的每一步处理都是相互独立的，所以编译器应该能够对其进行循环展开（Loop Unrolling）和循环体流水化（Loop Pipelining）。而且，由于循环内部完全避免了分支操作，所以处理器上的流水线处理也可以高效地执行。这些都可以提高 PForDelta 的解码速度[2]。当然，使用 1 个循环（1 个阶段）也可以完成解码处理，但是由于将其分为 2 个阶段后解码速度能够明显提升，所以还是使用了 2 个循环。

> [2]虽然在第 2 个循环的处理中会发生数据冒险（Data Hazard），但是由于要填补的整数并不多，所以并不会引起较大的系统开销。

❙ 关于各种编码方法

在参考文献 6 中，Zhang 等人比较了各种倒排文件的编码方法（rice、variable-byte、S9 和 PForDelta）。观察比较结果之后，可以得出以下有关压缩率的结论。

rice ＞ PForDelta ＞ S9 ＞ variable-byte（越往左压缩率越高）

而关于解码速度，可以得到以下的结论。

PForDelta ＞ S9 ＞ variable-byte ＞ rice（越往左解码速度越快）

由于这些结论很大程度上依赖于进行实验的数据集，所以不妨只把它们当作是参考数据。而且，各种编码方法的好坏未必能够通过压缩率和解码速度比较出来。例如，虽然 variable-byte 编码的解码速度不如 PForDelta 和 S9，但是 variable-byte 却拥有易于实现、可进行增量编码（Incremental Encoding）的优点。总之，我们应该在彻底了解各种编码方法的基础上，再根据搜索引擎的用途和运行搜索引擎的硬件性能，具体问题具体分析地选用这些编码方法。

动态索引构建

在 1-7 小节中我们曾经讲过，在构建索引的方法中有“静态”和“动态”之分，而且很多案例都需要动态的索引构建。动态索引构建是一种一边使索引结构时刻保持在可检索的状态，一边构建索引的方法。只要搜索引擎支持动态索引构建，那么新文档一添加，该文档的信息就会立即反映到索引上。也就是说，索引会时刻保持在最新的状态上。但是，为了使索引时刻保持在最新且可检索的状态上，就不得不在某种程度上容忍检索处理性能的下降。也就是说，“动态索引构建”和“高速检索处理”无法兼得。

在本节，我们将讲解几个具有代表性的动态索引构建方法，并观察这些方法是如何权衡动态构建和高速检索处理的。另外，关于这种动态索引构建的讨论一直以来都是以磁盘驱动器为二级存储器广泛展开的，所以在本节，我们还以磁盘驱动器（以下简称为磁盘）为对象进行讲解。

▍用于动态索引构建的基本策略

动态索引构建的基本策略如下所示。

- 将索引分成内存上的索引和磁盘上的索引并分别管理
- 添加文档后，优先更新内存上的索引
- 只有当内存上的索引大小达到了（事先设定好的）内存容量的上限时，才将其整合到磁盘上的索引中

动态索引构建的要点在于如何整合内存上的索引和磁盘上的索引。下面就让我们来略微深入地看一看。

▍整合索引

整合索引的方法大体上可以分为如下两种。

- 基于原地更新的整合（**Inplace Index Maintenance**）
- 基于合并的整合（**Merge-based Index Maintenance**）

在整合时，基于原地更新的整合是一种通过将内存上索引的倒排列表添加到磁盘上索引的倒排列表中，以尽可能缩小磁盘上索引更新范围的策略。具体来讲就是，事先在各倒排列表中预留出多余的空间，然后再将

内存上索引的倒排项存储到该空间中。当预留出的空间不足时，就将该倒排列表移动到其他的空间。

这既是广泛应用于数据库管理系统（DBMS）的数据管理方法，也是以往动态索引构建的主要方法。但是，由于在整合处理中要对磁盘进行大量的随机访问，所以近几年以磁盘扫描为基础的基于合并的整合方法渐渐成为了主流。

基于合并的整合

与基于原地更新的整合不同，基于合并的整合方法并不会直接更新磁盘上的索引，而是会对磁盘上已有的索引和内存上的索引进行合并，并将合并后的索引写入到一块新的磁盘空间（一个新的文件）中。下面我们将介绍 3 种在基于合并的整合方法中具有代表性的策略。

- 再合并策略

  再合并（Remerge/Immediate Merge）策略会通过不断地合并，始终在磁盘上保留 1 个索引。也就是说，当内存上索引的大小达到了某个阈值时，就需要将其和磁盘上的索引合并以生成新的索引（生成了新的索引以后，还要将作为合并源的磁盘上的索引和内存上的索引删除掉）。

  在这种策略中，获取 1 个倒排列表所需的磁盘寻道次数为 1 次，但是每次合并都要对磁盘上的索引进行全扫描。

  设 n 为要构建的倒排项的总数，b 为内存上能够存储的倒排项的个数，那么对于 n 个倒排项就要进行 n/b 次的合并，因此在构建时，磁盘操作的时间复杂度为 $O(n^2/b)$。

- 不合并策略

  不合并（No Merge）策略正如其名，是一种完全不进行合并的策略。当内存上的索引大小达到了某个阈值时，就将该内存上的索引写入到一个新的磁盘文件中。

  由于这种策略并不合并索引，所以虽然索引构建的性能较高，但是磁盘上的索引却被分散到了多个文件中，这就会导致在获取倒排列

表时，磁盘寻道的次数会与索引的数量成比例地增长。由于构建 n 个倒排项要进行 1 次合并，所以构建时磁盘操作的时间复杂度为 $O(n)$。

- **Geometric Partitioning** 策略

  Geometric Partitioning 策略是处于上述两种合并策略之间的一种方法。

  该策略会将磁盘上的索引分割成大小不同的多个索引片段，然后始终通过将内存上的索引与磁盘上较小的索引片段进行合并来削减合并过程中的输入输出量。

  而且，通过阶段性地将磁盘上的索引片段与更大的索引片段合并，可以在保存磁盘上短小索引片段的同时，控制磁盘上索引的个数。具体做法就是，先引入一个参数 r（一般令 $r = 2$ 或 $r = 3$），然后将 0 个或大于 $r^{(k-1)b}$ 且小于 $(r-1)r^{(k-1)b}$ 个倒排项，存储到按不同大小划分出来的第 k 级（$k = 1, 2, 3...$）索引片段中。

  此时，在内存上构建出的索引通常要与与第 1 级索引片段进行合并。然后，当倒排项的个数超过了第 1 级的范围（上限）时，就要将第 1 级索引片段与上一级的索引片段进行合并。以此类推，反复进行合并处理，直到各级别索引片段中的倒排项的个数均小于该级别的上限 [3]。

  下面，让我们来看一下当 $r = 2$ 时的具体合并过程。当 $r = 2$ 时，可以通过与二进制数加一相同的过程进行合并处理。也就是说，二进制数的各位就相当于级别，进位就相当于与上一级的索引片段进行合并处理。例如，可以用 00000100 表示在第 3 级上有索引片段。

  将内存上的索引依次合并到该状态，就会得到如下结果。

  00000100 → 00000101 → 00000111 → 00001000 → ……

  当 $r = 2$ 时，由于 n 个倒排项需要进行 $\log_2(n/b)$ 次合并，所以构建时磁盘写操作的时间复杂度为 $O(n \log_2(n/\mathrm{b}))$。

在 Geometric Partitioning 中，通过调整参数 *r*，就可以权衡索引的构建性能和检索性能。

- 当 ***r*** 较小时 → 构建性能优先
- 当 ***r*** 较大时 → 检索性能优先

图 A-2 中画出了在本小节介绍过的各种合并策略的构建过程。在各种策略中，构建时向磁盘写入时的时间复杂度和磁盘上的索引数，如表 A-1 所示。另外，有关这些方法的详细内容，请阅读参考文献 7。

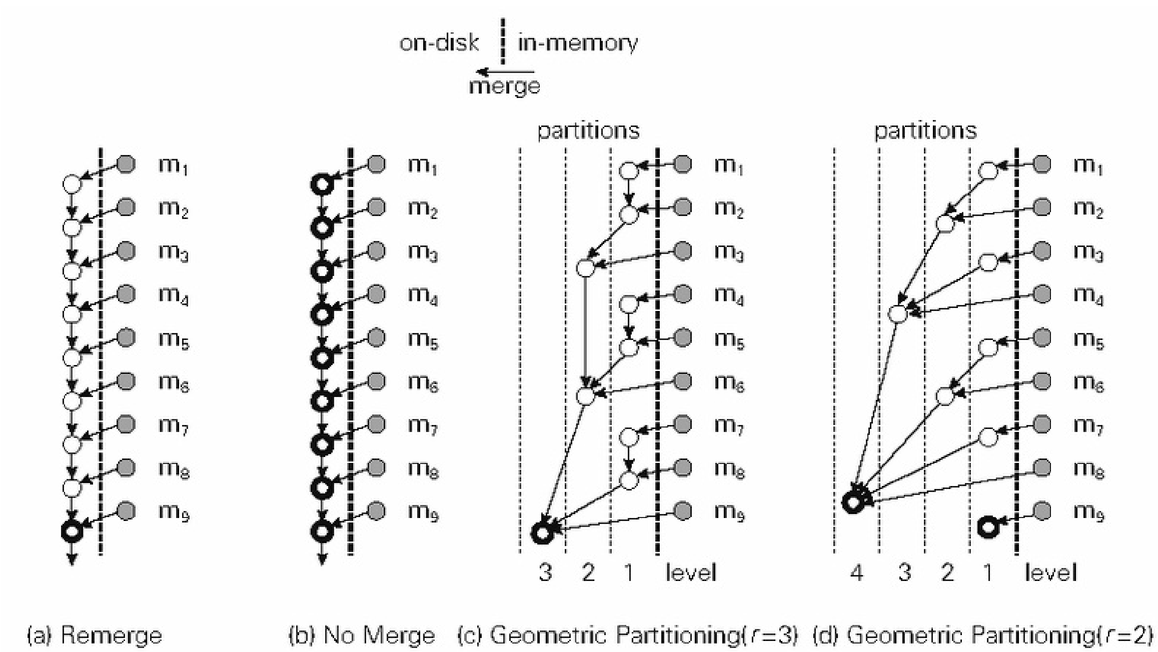


图 **A-2**　各种合并策略的构建过程

※带颜色的节点 mi 表示在内存上构建的第 i 个索引片段。线条加粗的节点表示将 9 个内存上的索引与磁盘上的索引合并后的索引片段。

表 **A-1**　各种合并策略的性能特性

	Remerge	**No Merge**	**Geometric Partitioning（*r* = 2）**
构建时磁盘操作的时间复杂度	$O(n^2/b)$	$O(n)$	$O(n\log(n/b))$

磁盘上的最大索引数	1	n/b	$\log_2(n/b)$

> [3]当在多个级别上发生连续的索引片段合并时，通常都是进行多路合并，而不是阶段性地两两合并。

分布式索引

▍将倒排索引分布到多台计算机上的方法

当需要检索那些为大量文档构建的索引或需要处理大量查询时，1 台计算机未必具备足够的性能。为此，在大规模的搜索引擎中，为了提高检索处理的性能，通常都会将索引分布到多台计算机上。据说在像 Google 等 Web 搜索引擎中，早在 2003 年左右，就已经将几十 TB 规模的索引分布到几千台服务器上了（参考文献 8）。

使用多台计算机提高检索处理性能的方法大体上可以分为以下两种。

- 复制索引（**Replication**）
- 分割索引（**Partitioning**）

所谓复制索引，就是将相同的倒排索引（的副本）配备到多台计算机上。在检索时，一旦将查询发送给中继服务器（Receptionist），中继服务器就会将查询再转发到其中的某一台计算机上，并由这台计算机来完成检索处理。

由于这种方法能够使吞吐量随着计算机台数的提升而提升，所以当查询的吞吐量至关重要时，该方法将非常有效。虽然复制索引具有通过中继服务器进行检索的特点，但是在检索处理的步骤上该方法并没有什么亮点，所以对它的讲解先告一段落。

另一方面，所谓分割索引，就是将倒排索引分割成多份，然后将分割而成的倒排索引片段配备到多台计算机上的方法。而在检索时，一般都需要询问多台计算机，然后再把得到的多条结果整合起来。一般认为该方法适用于想要缩短查询响应时间的场景。下面，我们就来进一步讲解分割索引的方法。

▍两种分割倒排索引的方法

分割索引的方法大体上可以分为两种。

- 按文档划分

  按文档划分（Document Partitioning）是一种对文档集合进行分割后，将分割后的文档集合连同相应的倒排索引分布到多台计算机（以下称为索引服务器）上的方法。如图 **A-3** 所示，该方法其实就是对倒排索引进行了纵向分割。

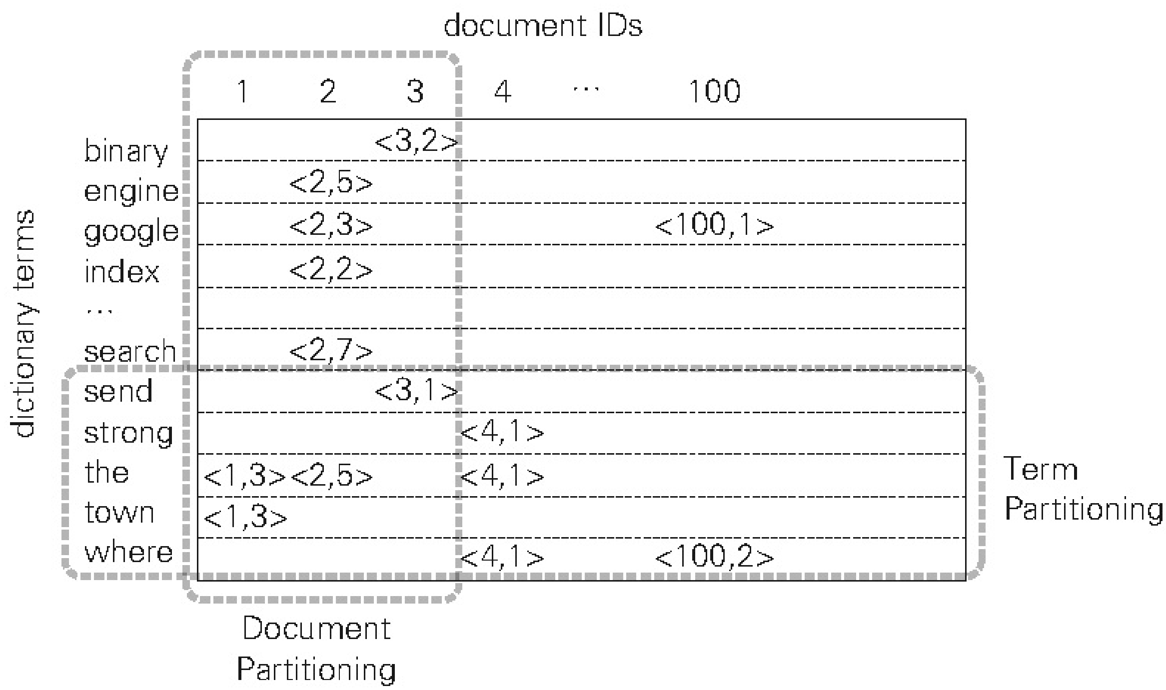
  

  图 **A-3**　按文档划分其实是对倒排索引进行了纵向分割

  在检索时，由中继服务器将查询发送到全部的索引服务器上。各索引服务器只对自己管理的索引进行检索处理，并将检索结果返回给中继服务器。中继服务器将各结果合并后，再将最终结果返回给请求者。

  中继服务器的整合处理主要由以下两个步骤构成。

  - 合并检索结果

  - 重新排名

例如，当请求者需要获取前 *k* 个按照查询与文档的关联度排序的文档时，首先就要从各索引服务器获取前 *k* 个按关联度排序的文档的文档信息（文档编号、得分等）。然后，将这些结果合并成一个整体，计算出排在前 *k* 个的文档编号。最后，将文档编号连同对应的文档信息一起作为结果返回给请求者。

文档信息有时是存储在索引服务器上的，也有时会集中存储在其他服务器上。按文档划分的示意图如图 A-4 所示。

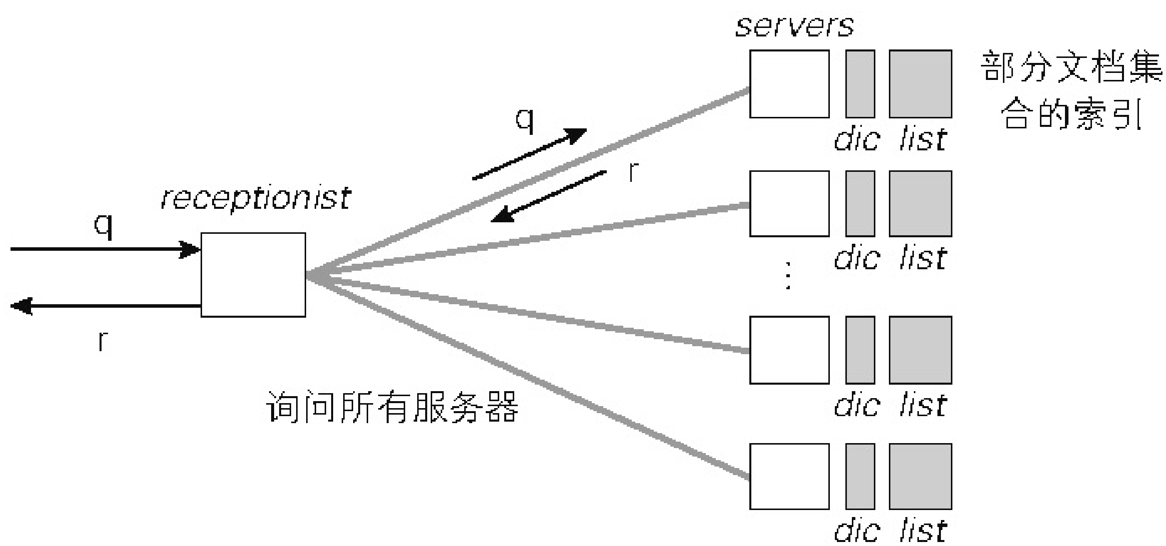


图 **A-4**　按文档划分的示意图

- 按单词划分

按单词划分（Term Partitioning）是一种对词典进行分割后，将分割后的词典片段连同这一部分词典中的单词所对应的倒排索引以及文档集合分布在多台计算机中的方法。如图 A-3 所示，该方法其实就是对倒排索引进行了横向分割。

在检索时，首先由中继服务器根据单词和索引服务器的对应表，将要检索的词语发送到相应的索引服务器上，并由这些索引服务器完成检索词的检索处理。然后，索引服务器再将作为检索结果的倒排文件返回给中继服务器，并在中继服务器上进行多个倒排文件的合并处理等操作。最后与按文档分割时相同，最后再将一部分合并后的结果返回给请求者。按单词划分的示意图如图 A-5 所示。

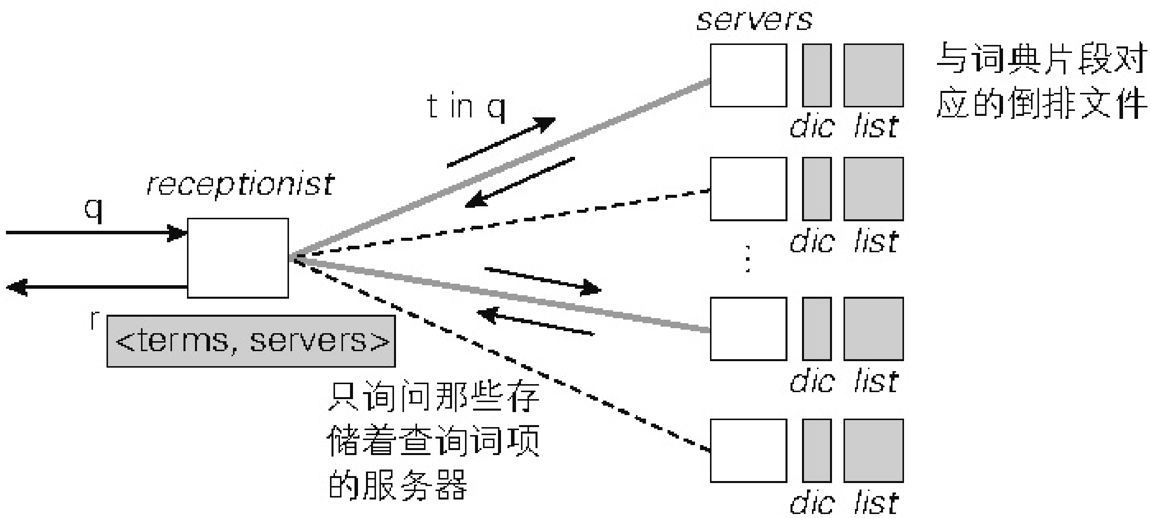


图 **A-5**　按单词划分的示意图

这两种分割方式各有利弊，下面我们就来定性地比较一下它们[4]。

在按文档划分中，由于分割的是文档集合，所以各索引服务器中都分配到了大小差不多的索引和文档集合。因此，该方法具有检索时各索引服务器的负载都很平均的特点。也就是说，可以通过增加计算机的数量来加快检索处理的速度。这种较高的可扩展性使得按文档划分广泛应用于以 Google 和 Yahoo! 为代表的大规模搜索引擎中。

另一方面，对于按单词划分，由于分割的是词典，而且每个单词在文档内的出现频率又不相同，所以由各索引服务器管理的倒排索引的大小以及文档集合的大小都有可能很不平均。而且，在检索时，由于只询问了管理检索词的索引服务器，而检索词的检索频率又不相同，所以这可能会导致对索引服务器的询问次数分布不均的情况。因此，就会出现各索引服务器负载不平均，部分索引服务器负载较高的情况。

但是，从在检索时要从二级存储器读取数据的角度来看，按单词划分还是具有一些优势的。在按单词划分时，由于各检索词所对应的倒排列表都在 1 台索引服务器上，所以只需要进行 1 次顺序访问即可获取所需的倒排列表。而另一方面，在按文档划分时，为了获取各检索词所对应的倒排列表，需要对每台索引服务器都进行 1 次顺序访问。也就是说，相对于按文档划分，按单词划分具有输入输出

的总数较少的优点。

[4]定量比较的内容请阅读参考文献 9。

A-2　wiser 中的文本提取和存储

用于处理 XML 的 2 种 API——DOM 和 SAX

由于第 2 章并没有详细地讲解 wiser 中的“文本提取器”，所以在本节，就让我们深入地了解一下在该模块上进行的处理吧。

由于本书所使用的 Wikipedia 的词条数据全部来源于 1 个巨大的 XML 文件，所以要想通过 wiser 构建索引，就需要先从包含了所有词条的 XML 文件中提取出各个词条（文档），然后再从各个词条中提取出标题和正文。

为了加载 XML 文件，我们在 wiser 中使用了名为 expat 的代码库。用于处理 XML 的 API 分为以下两种。

- **DOM（Document Object Model）**
- **SAX（Simple API for XML）**

使用 DOM 时，会先将整个 XML 文档全部加载到内存中，然后再开始解析处理。也就是说，这种方法的优点在于操作时可以忽略元素的顺序，而缺点在于会消耗大量的内存。虽然 DOM 使用起来很方便，但是会占用大量的资源。与此相反，在使用 SAX 时，由于是一边加载 XML 中的元素，一边依次进行处理的，所以只需少量的内存即可完成处理，但是处理时不得不考虑 XML 文档中元素的顺序。综上所述，DOM 和 SAX 各有各的优缺点。

由于包含着 Wikipedia 词条的 XML 文件相对来说还是比较大的。所以对于现在（2014 年）的个人计算机而言，要把所有的数据都加载到内存上恐怕还有些困难。因此，在 wiser 中我们会使用 SAX 来处理 XML。

提取文档的标题和正文

在 wiser 中，有关从 Wikipedia 词条数据中提取文档的处理过程都写在了文件 wikiload.c 中的函数 load_wikipedia_dump() 中。由于该函数中并

没有进行十分复杂的处理，所以我们就一行一行地往下读吧。

简单来说，函数 load_wikipedia_dump() 的作用是用 SAX 解析含有 Wikipedia 词条数据的 XML 文档，并从中提取出词条的标题和正文。提取出的标题和正文会通过存储文档的回调函数存储到数据库中，而该回调函数会作为参数传入函数 load_wikipedia_dump() 中。下面我们就来梳理函数 load_wikipedia_dump() 的源代码。

```
/**
 * 加载Wikipedia的副本（XML文件），并将其内容传递给指定的函数。
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] path Wikipedia副本的路径
 * @param[in] func 接收env，词条标题，词条正文3个参数的回调函数
 * @param[in] max_article_count 最多加载多少个词条
 * @retval 0 成功
 * @retval 1 申请内存失败
 * @retval 2 打开文件失败
 * @retval 3 加载文件失败
 * @retval 4 解析XML文件失败
 */
int
load_wikipedia_dump(wiser_env *env,
                   const char *path, add_document_callback func, int max_
article_count)
{
  FILE *fp;
  int rc = 0;
  XML_Parser xp;
  char buffer[LOAD_BUFFER_SIZE];
  wikipedia_parser wp = {❶
    env,               /* 存储着应用程序运行环境的结构体 */
    IN_DOCUMENT,       /* 初始状态 */
    NULL,              /* 词条标题的临时存储区 */
    NULL,              /* 词条正文的临时存储区 */
    0,                 /* 初始化经过解析的词条总数 */
    max_article_count, /* 最多要解析多少个词条 */
    func               /* 将解析后的文档传递给该函数 */
  };

  if (!(xp = XML_ParserCreate("UTF-8"))) {❷
    print_error("cannot allocate memory for parser.");
    return 1;
  }

  if (!(fp = fopen(path, "rb"))) {❸
```

```
    print_error("cannot open wikipedia dump xml file(%s).",
                strerror(errno));
    rc = 2;
    goto exit;
  }

  XML_SetElementHandler(xp, start, end);❹
  XML_SetCharacterDataHandler(xp, element_data); ❹
  XML_SetUserData(xp, (void *)&wp);❺

  while (1) {❻
    int buffer_len, done;

    buffer_len = (int)fread(buffer, 1, LOAD_BUFFER_SIZE, fp);❼
    if (ferror(fp)) {
      print_error("wikipedia dump xml file read error.");
      rc = 3;
      goto exit;
    }
    done = feof(fp);

    if (XML_Parse(xp, buffer, buffer_len, done) == XML_STATUS_ERROR) {
      print_error("wikipedia dump xml file parse error.");
      rc = 4;
      goto exit;
    }

    if (done || (max_article_count >= 0 &&
                 max_article_count <= wp.article_count)) { break; }
  }
exit:
  if (fp) {
    fclose(fp);
  }
  if (wp.title) {
    utstring_free(wp.title);
  }
  if (wp.body) {
    utstring_free(wp.body);
  }
  XML_ParserFree(xp);
  return rc;
}
```

在 wiser 中，我们将各种各样的信息都存储到了一个名为 wiser_env 的、作为全局变量使用的结构体中。这是一个作为全局变量使用的结构

体。wiser_env 的定义在文件 wiser.h 中。

函数 load_wikipedia_dump() 可接收 4 个参数，依次是刚刚提到的 wiser_env、Wikipedia 副本的路径、回调函数和最多加载多少个词条即最多为多少个词条创建索引。一旦打开了指定路径下的 XML 文件，就可以开始解析了。解析时每取出 1 个词条，就要调用 1 次录入词条的回调函数。

```
typedef struct {
  wiser_env *env;                /* 存储着应用程序运行环境的结构体 */
  wikipedia_status status;       /* 正在读取词条XML标签的哪一部分 */
  UT_string *title;              /* 词条标题的临时存储区 */
  UT_string *body;               /* 词条正文的临时存储区 */
  int article_count;             /* 经过解析的词条总数 */
  int max_article_count;         /* 最多要解析多少个词条 */
  add_document_callback func;    /* 将解析后的文档传递给该函数 */
} wikipedia_parser;
```

首先，在步骤❶中，我们初始化了一个 wikipedia_parser 类型的变量，用于管理解析 XML 时的状态和环境等。另外，为了管理作为文档标题（title）和正文（body）的字符串，我们使用了专门用于处理字符串的代码库 utstring[5]。

> [5]http://troydhanson.github.io/uthash/utstring.html

在❷的步骤中，为了使用作为 XML 解析器的 expat，我们进行了一些准备工作。这里仅仅是用 expat 创建了一个以字符编码 UTF-8 为目标编码的解析器。

在❸的步骤中，我们打开了给定路径下的 XML 文件。为了慎重起见，将 b 添加到了函数 fopen( ) 的第二个参数中，表示以二进制文件的模式读取文件。之所以这样做，是因为如果作为文本文件处理，就难免要进行一些额外的转换工作。

另外，我们还必须将“在这个时候要这样做”的处理过程注册到刚刚生成的 XML 解析器上。在 SAX 中，可以根据 XML 的标签设定要调用的函数。

在❹的步骤中，用函数 XML_SetElementHandler() 注册了遇到起始标签

或结束标签时要调用的函数。在这里，我们创建并注册了函数 start() 和 end()。

同样，接下来又设定了遇到标签中的字符串时，要调用函数 element_data()。

在❺中设定的是将刚刚提及的 wikipedia_parser 类型的变量 wp 的指针传递给各回调函数。设定好以后，我们就可以通过 wp 在函数之间进行信息交换了。

❻是读取文件内容后，调用 SAX 的 API——函数 XML_Parse() 的循环。在该循环中，刚刚注册的函数 start()、end() 和 element_data() 会被调用。有关该循环的细节我们将在稍后讲解。

另外，在❼的步骤中，为了控制内存的使用量，我们每次只从 XML 文件中读取由常数 LOAD_BUFFER_SIZE 指定的字节数，并将这一部分数据加载到内存上。

掌握状态的迁移

下面，让我们再来看一下 start()、end() 和 element_data() 这 3 个函数。

在这里，请诸位注意变量 wp 中的成员变量 wp.status。wp.status 用于管理解析 XML 时的状态，在对 wp 进行初始化时，我们将 status 字段的值设为了 IN_DOCUMENT。随着 XML 文件的不断加载，我们还要不断地变更 wp.status 的取值。

❙ 函数 **start()** 和函数 **end()**

首先，我们来看一下遇到起始标签时会被调用的函数 start() 和遇到结束标签时会被调用的函数 end()。这两个函数的职责是根据标签种类的不同，相应地变更状态。

```
/**
 * 遇到XML的起始标签时被调用的函数
 * @param[in] user_data Wikipedia解析器的运行环境
 * @param[in] el XML标签的名字
 * @param[in] attr XML标签的属性列表
 */
```

```
static void XMLCALL
start(void *user_data, const XML_Char *el, const XML_Char *attr[])
{
  wikipedia_parser *p = (wikipedia_parser *)user_data; ❽
  switch (p->status) { ❾
  case IN_DOCUMENT:
    if (!strcmp(el, "page")) {
      p->status = IN_PAGE;
    }
    break;
  case IN_PAGE:
    if (!strcmp(el, "title")) {
      p->status = IN_PAGE_TITLE;
      utstring_new(p->title);
    } else if (!strcmp(el, "id")) {
      p->status = IN_PAGE_ID;
    } else if (!strcmp(el, "revision")) {
      p->status = IN_PAGE_REVISION;
    }
    break;
  case IN_PAGE_TITLE:
  case IN_PAGE_ID:
    break;
  case IN_PAGE_REVISION:
    if (!strcmp(el, "text")) {
      p->status = IN_PAGE_REVISION_TEXT;
      utstring_new(p->body);
    }
    break;
  case IN_PAGE_REVISION_TEXT:
    break;
  }
}

/**
 * 遇到XML的结束标签时被调用的函数
 * @param[in] user_data Wikipedia解析器的运行环境
 * @param[in] el XML标签的名字
 */
static void XMLCALL
end(void *user_data, const XML_Char *el)
{
  wikipedia_parser *p = (wikipedia_parser *)user_data; ❽
  switch (p->status) { ❿
  case IN_DOCUMENT:
    break;
  case IN_PAGE:
```

```
    if (!strcmp(el, "page")) {
      p->status = IN_DOCUMENT;
    }
    break;
  case IN_PAGE_TITLE:
    if (!strcmp(el, "title")) {
      p->status = IN_PAGE;
    }
    break;
  case IN_PAGE_ID:
    if (!strcmp(el, "id")) {
      p->status = IN_PAGE;
    }
    break;
  case IN_PAGE_REVISION:
    if (!strcmp(el, "revision")) {
      p->status = IN_PAGE;
    }
    break;
  case IN_PAGE_REVISION_TEXT: ⓫
    if (!strcmp(el, "text")) {
      p->status = IN_PAGE_REVISION;
      if (p->max_article_count < 0 ||
          p->article_count < p->max_article_count) {
        p->func(p->env, utstring_body(p->title), utstring_body(p->body));
      }
      utstring_free(p->title);
      utstring_free(p->body);
      p->title = NULL;
      p->body = NULL;
      p->article_count++;
    }
    break;
  }
}
```

首先，在❽的步骤中，我们将变量 user_data 的类型转换成了 wikipedia_parser 的指针。这样做是为了可以通过变量名来访问 wikipedia_parser 类型的成员。

在❾的 switch 语句中，根据状态 p->status 的取值，产生了若干个分支，分别处理各个状态。

- 当状态为 **IN_DOCUMENT** 时

→如果遇到了 **page** 标签，就将状态变更为 **IN_PAGE**，表明当前解析到了 **page** 标签中

- 当状态为 **IN_PAGE** 时

  →如果遇到了 **title** 标签，就将状态变更为 **IN_PAGE_TITLE**，表明当前解析到了 **title** 标签中

  →如果遇到了 **revision** 标签，就将状态变更为 **IN_PAGE_REVISION**，表明当前解析到了 **revision** 标签中

- 当状态为 **IN_PAGE_REVISION** 时

  →如果遇到了 **text** 标签，就将状态变更为 **IN_PAGE_REVISION_TEXT**，表明当前解析到了 **text** 标签中

同样地，在❿的 switch 语句中，我们还是根据状态产生分支并分别处理各个状态。

- 当状态为 **IN_PAGE** 时

  →如果遇到了 **page** 的结束标签，就将状态变更为 **IN_DOCUMENT**

- 当状态为 **IN_PAGE_TITLE** 时

  →如果遇到了 **title** 的结束标签，就将状态变更为 **IN_PAGE**

- 当状态为 **IN_PAGE_REVISION** 时

  →如果遇到了 **revision** 的结束标签，就将状态变更为 **IN_PAGE**

- 当状态为 **IN_PAGE_REVISION_TEXT** 时

  →如果遇到了 **text** 的结束标签，就将状态变更为 **IN_PAGE_REVISION**

  →与此同时，以刚刚取出的标题和正文作为参数，调用回调函数

通过结合函数 start() 和函数 end() 的处理过程，就可以管理与正在解析的 XML 的位置相对应的状态了。相对于函数 start()，在函数 end() 中进行的处理稍显复杂，有关状态迁移以外的部分我们将在稍后讲解。

解析处理的状态迁移如图 A-6 所示。

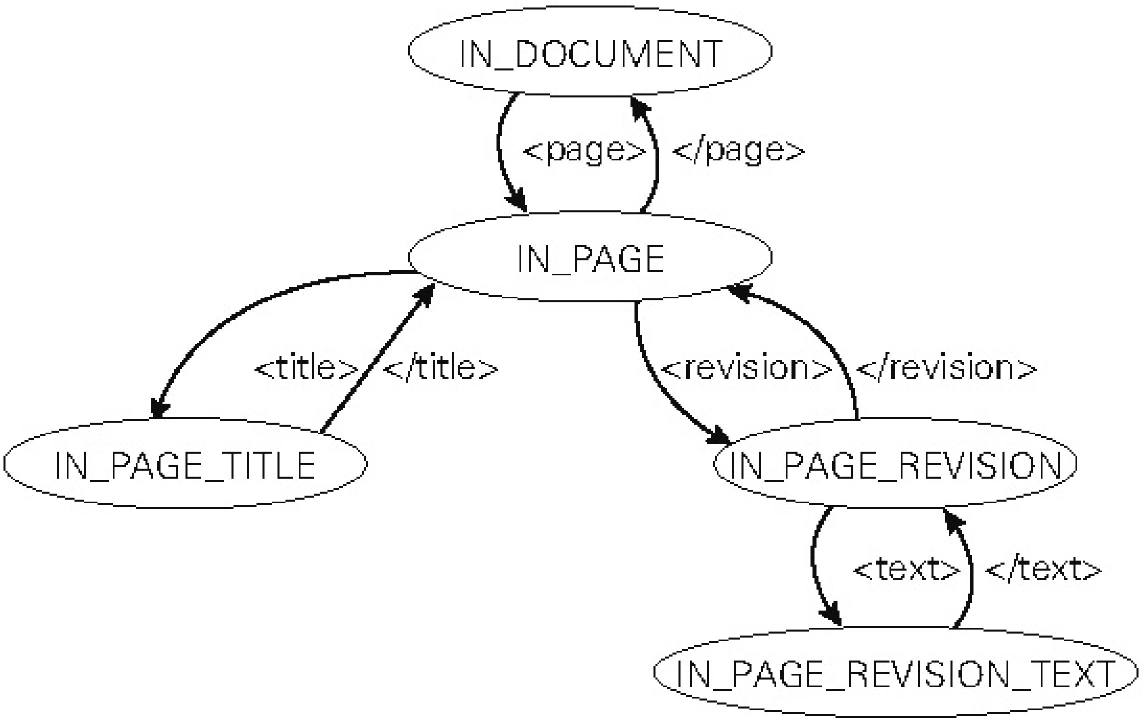


图 **A-6　wikiload.c** 中的状态迁移

❙ 函数 **element_data()**

下面，让我们再来看一下遇到标签中的字符串时会被调用的函数 element_data()。该函数的作用是获取 XML 文档中的文本。通过函数 start() 和 end() 设定的状态会在该函数中发挥作用。

```
/**
 * 解析XML元素的数据时被调用的函数
 * @param[in] user_data Wikipedia解析器的运行环境
 * @param[in] data 元素中的数据
 * @param[in] data_size 数据的大小
 */
static void XMLCALL
element_data(void *user_data, const XML_Char *data, int data_size)
{
```

```
  wikipedia_parser *p = (wikipedia_parser *)user_data;
  switch (p->status) {
  case IN_PAGE_TITLE:
    utstring_bincpy(p->title, data, data_size);
    break;
  case IN_PAGE_REVISION_TEXT:
    utstring_bincpy(p->body, data, data_size);
    break;
  default:
    /* do nothing */
    break;
  }
}
```

在函数 element_data() 中，我们会根据状态进行如下的处理。

当状态为 **IN_PAGE_TITLE** 时

→将字符串作为页面标题存储

当状态为 **IN_PAGE_REVISION_TEXT** 时

→将字符串作为页面正文存储

也就是说，要根据状态来判断由 XML 解析器传递过来的字符串是标题还是词条正文，或是其他的什么数据。前面拼命地更新状态实际上全是为了函数 element_data()。

▍函数 **end()** 所做的处理

既然已经明白了状态的作用，我们就再来回顾一下函数 end()。请诸位注意当遇到 </text> 标签时进行的处理⓫。

首先，当存储着 Wikipedia 词条正文的 <text> 标签结束时，我们会调用将词条存储到数据库中的回调函数。该回调函数存储在用户数据（变量 p）中。此时，一旦已存储的词条总数超过了词条的最大索引数，就不再调用该回调函数了。

构建文档数据库

为了存储从 XML 词条数据中提取出的标题和正文数据，我们在 wiser 中使用 SQLite 这种 RDBMS（关系型数据库管理系统）。

下面我们就实际接触一下 SQLite，看看 wiser 是如何存储文本数据的吧。首先，通过以下的命令，即可以会话模式启动 SQLite。在有些环境执行时，需要用名为 sqlite3 的命令代替 sqlite 命令。

示例

```
> sqlite test.db
SQLite version 3.7.11 2012-03-20 11:35:50
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

▍创建表

在 RDBMS 中，记录的集合会被存储到名为表的结构中。表中的每条记录都带有若干个属性（列），数据就存储在各个属性中。而每条记录由什么样的属性构成则需要事先设定。

在这里，我们创建了一张如下所示的用于存储文档的表。

示例

```
CREATE TABLE documents (
  id    INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  body  TEXT NOT NULL
);
```

▍确认是否正确地创建了数据库

我们可以通过下面几条 SQLite 命令来确认是否正确地创建了数据库。

- **.tables** 命令：输出数据库中的表名列表
- **.schema** 命令：输出指定表的定义

示例

```
sqlite> .tables
documents
sqlite> .schema documents
CREATE TABLE documents
  id    INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  body  TEXT NOT NULL
);
```

上述命令输出的字符串其含义如下所示。

- **documents**：表名
- **id**、**title**、**body**：表中的列名
- **INTEGER**、**TEXT**：列的类型（**INTEGER** 表示数值类型、**TEXT** 表示字符串类型）
- **PRIMARY KEY**：指定能够唯一确定表中各行的列（主键）
- **NOT NULL**：指明该列不接受表示空值的 **NULL**

为了易于用 C 语言处理，我们在这里将数值类型的 id 作为了主键，而未使用字符串类型的 title。

▌添加新的记录

要向 RDBMS 中添加新的记录时，需要像下面这样使用 INSERT INTO 语句。

示例

```
sqlite> INSERT INTO documents (title, body)
  VALUES ('test', 'This is a test document.');
sqlite> INSERT INTO documents (title, body)
  VALUES ('sample', 'This is a sample document.');
```

在 INSERT INTO 语句中，需要先指定作为添加目标的表名。然后，再对 VALUES 关键字设定以下 2 个信息。

- 列名的列表
- 与前面各列相对应的值的列表

在 SQLite 中，被指定为 INTEGER PRIMARY KEY 的列会被自动地填充数值。因此，无需明确地为该列设定值。

▍查看已存储的记录的列表

我们可以像下面这样，使用 SELECT 语句列出存储在 RDBMS 表中的记录。

示例

```
sqlite> SELECT * FROM documents;
1|test|This is a test document.
2|sample|This is a sample document.
```

在 SELECT 语句中，还可以像下面这样，只将满足特定条件的记录列出来。

示例

```
sqlite> SELECT * FROM documents WHERE title = 'test';
1|test|This is a test document.
```

像上面这样，就只会显示 title 字段为字符串 test 的记录了。

▍更新数据

我们还可以使用 UPDATE 语句的查询来更新数据。

示例

```
sqlite> UPDATE documents SET title = 'changed title' WHERE id = 1;
```

```
sqlite> SELECT * FROM documents;
1|changed title|This is a test document.
2|sample|This is a sample document.
```

在 wiser 的 database.c 中的函数 db_add_document() 中，我们就是通过上述几种查询语句，将 Wikipedia 词条的标题和正文存储到数据库中的。

至此为止，我们就讲解完了 wiser 是如何从 Wikipedia 的 XML 词条数据中提取标题和正文，以及又是如何将它们存储起来的。虽然通过 SAX 解析 XML 文档有些麻烦，但是这里我们所做的处理却很简单，只是从 XML 中将文档的标题和正文提取出来后存储到文档数据库中而已。

后记

为了开发实用的搜索引擎，首先必须要尝试去开发。就请诸位使用优秀的开源检索库，先试着搭建一个简单的检索服务吧。

然后，再将这个检索服务公诸于世。想必随后诸位就要开始面对一些未曾预料到的问题了。

用户输入了冗长的查询

用户通过程序定期提交搜索表单以获取信息

用户意图挖掘检索服务的漏洞

只是简单地想想诸如此类的问题，诸位就应该能意识到，本书未曾提及的问题会层出不穷。

让我们通过技术水平和服务品质的提升来解决这些问题吧。只要以本书的知识为基础，再经过反复的实践，诸位的检索服务定会稳健起来。

笔者之所以推荐使用开源的检索库，是因为在优化服务的过程中，还需要对检索库加以修改。而在修改时，本书所介绍的有关 wiser 的知识就会发挥作用。如果诸位所进行的修改能使很多人受益，那么就请把这个修改反馈给检索库的创建者吧。这样一来，凡是使用了该检索库的检索服务都会得到升华。

笔者欣然期盼有朝一日能够使用上由诸位读者开发的检索服务，并希望本书能对诸位有所帮助。

末永匡

于 2014 年 8 月

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
